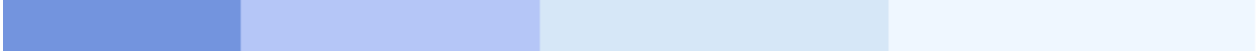




SMARTS

HTTP Server



This document applies to Complete Version 6.2.1 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

© March 2002, Software AG
All rights reserved

Software AG and/or all Software AG products are either trademarks or registered trademarks of Software AG. Other products and company names mentioned herein may be the trademarks of their respective owners.

Table of Contents

SMARTS HTTP Server	1
SMARTS HTTP Server	1
Introduction to the HTTP Server	2
Introduction to the HTTP Server	2
The HTTP Server	2
SMARTS Implementation of the Common Gateway Interface (CGI)	3
SMARTS CGI Input Processing	3
SMARTS CGI Output Processing	3
SMARTS CGI Environment Variables	5
SMARTS CGI Termination Processing	6
SMARTS CGI Extensions	6
Standard CGI Operation	6
Non-C CGI Programs	7
Extensions for Other Languages	7
The Conversational CGI Program Concept	7
The HTTP Server Solution	8
Conversational CGI Application Structure	8
General Installation Information	10
General Installation Information	10
Supported Operating Systems	10
Supported Environments	10
Installation Media	11
Installation Overview	11
Installation on OS/390 or MVS	12
Installation on OS/390 or MVS	12
The Installation Tape	12
Tape Contents	12
Creating the PC Files	13
Creating the Mainframe Datasets	13
Running the Installation Jobs	15
Installing under the SMARTS Server Environment	16
Installing under Com-plete	17
Where Next ?	18
Installation on VSE/ESA	19
Installation on VSE/ESA	19
The Installation Tape	19
Tape Contents	19
Running the Installation Jobs	20
Installing under the SMARTS Server Environment	21
Installing under Com-plete	21
Where Next ?	22
Verifying the Installation	23
Verifying the Installation	23
Verify Operation of the Servers	23
Sample Programs	23
Sample HTML Files	23
Verify the SMARTS HTTP Server Installation	24

Prepare the Sample Programs	24
Start the HTTP Server	24
Troubleshooting	24
The HTTP Server Initialization Fails	25
The SMARTS Environment Initializes, but the HTTP Server Initialization Fails	25
All SMARTS Components Initialize, but Access Attempts Fail	25
Customizing and Using the HTTP Server	26
Customizing and Using the HTTP Server	26
Initializing the HTTP Server	26
Termination	27
Operator Commands	27
Configuration	28
Sample HAANCONF Member	28
Assembling the Configuration Member	28
HTTP Server Parameters	29
CGIPARM	29
CONBUFL	29
CONV	29
DEFACEE	29
DFLTCONT	30
DFLTURL	30
HTTPUSER	30
HTTPLIST	31
HTTPHCD	31
LOGON	31
MSGCASE	31
NATLIB	32
NATPARM	32
NATTHRD	32
PORT	32
RECVBUFL	32
SEND	33
SENDBUFL	33
SERVNAME	33
SERVPOOL	34
TRACE	34
TRACEDD	34
URLPBUFL	35
Content Processing	35
Member Type Processing	35
Dataset Name Processing	36
CGI Request Output Processing	36
Configurable Tables	37
HAANEUTT	37
HAANIPTT	37
HAANIUTT	37
HAANOPTT	37
HAANTOTT	37
Default URL Processing	37
Resource Usage	38
Global Storage	38

HAANLIST Storage	38
HAANRQST Storage	38
Additional Storage Used for CGI Requests	39
Pooled Server Processing	39
Advantages of Pooled Servers	39
Considerations when Using Pooled Servers	40
Natural Considerations	40
Conversational Processing	40
Installing NATURAL CGI	41
Installing NATURAL CGI	41
Natural 2.2 Support	41
Natural Tasks	41
Relationship to the HTTP Server Configuration	42
Invoking a Natural CGI Program	42
Installation Verification	43
Additional Notes	43
Using the Natural Web Interface	44
Required Tasks	44
Invoking a Natural Web Interface Program	44
Installation Verification	44
Additional Notes	45
Security	46
Security	46
The Default User	46
HTTP Server Security Integration	47
Logon Allowed (LOGON=ALLOWED)	47
Logon Required (LOGON=REQUIRED)	47
Logon Disallowed (LOGON=DISALLOWED)	47
HTTP User ID and Password Encryption	48
Natural Security Considerations	48
Implementing SAF Security	48
Programming CGI Requests	49
Programming CGI Requests	49
HAANUPR: The HTTP Server User Program Request Module	49
Standard Return and Reason Codes	49
The CONVERSE Function	50
The DISABLE-CONVERSE Function	51
The ENABLE-CONVERSE Function	51
The GET-DATA Function	52
The LIST-DATA Function	54
The PUT-BINARY Function	56
The PUT-TEXT Function	57
HAANCGIG Interface Module	58
HAANCGIL Interface Module	59
HAANCGIP and HAANCGIT Interface Modules	61
CGI Extension Interface Module Status	62
Interface Module Return Codes	62
Interface Module Reason Codes	63
Running CGI Programs under SMARTS	65
Running CGI Programs under SMARTS	65
The SMARTS Server Environment	65


Linking the Program	65
Requirement	66
The Com-plete Environment	66
Linking the Program for Com-plete	66
Preparing Com-plete for the Application	66
Calculating the Catalog Size under Com-plete	66
Catalog Size for CGI Programs under Com-plete	67
Program Index Entries under Com-plete	67
Running the Program under Com-plete	68
Program Options or Functions to Avoid under Com-plete	68
Recommendations for the Com-plete Environment	68
Recommendations for Cobol Running under Com-plete	69
Natural Considerations	69
Running Natural Applications	69
Natural and the SMARTS CGI Extensions	69
Natural Script	70
Additional Notes on Natural	72
C Considerations	73
Compiling and Linking C Applications	73
Supplied C Sample Programs and Jobs	73
SMARTS and stdin, stdout, and stderr	74
C and the SMARTS CGI Extensions	74
COBOL Considerations	74
Sample Programs and Jobs	74
COBOL and the SMARTS CGI Extensions	75
PL/1 Considerations	75
Sample Programs and Jobs	75
PL/1 and External Module Names	76
PL/1 and the SMARTS CGI Extensions	76
S/390 Assembler Considerations	77
Assembler and the SMARTS CGI Extensions	77
Support and Maintenance	78
Support and Maintenance	78
Reporting Problems	78
Problem Resolution	79
Thread Dump Diagnosis under Com-plete	79
HTTP Server Trace Facilities	79
Applying Maintenance	79
The HTTP Server User Exit	81
The HTTP Server User Exit	81
Installation	81
General Interface	81
Exit Parameter List	82
Entry/Exit Processing	82
SMARTS API	83
Exit Points	84
Initialization	84
Termination	84
URL Processing	85
Output Processing	85
Input Processing	87

Accept Processing	88
Messages and Codes	89
Messages and Codes	89
SMARTS HTTP ABEND Codes	90
Overview of Messages	90
HTTP Server Messages (APSHTP Prefix)	91
Overview of Messages	91
Sample Members	111
Sample Members	111
On the SMARTS Source Dataset	111
On the SMARTS HTTP JOBS Dataset	112
On the Natural Library HTPvrs	113
On the Natural INPL Update for SYSWEB	113







SMARTS HTTP Server

This documentation describes the HTTP server component of the Software AG Multiple Architecture Runtime System (SMARTS) environment. It includes the following documents:








General Information

	Introduction to the HTTP Server	Introduces you to the HTTP Server, including the CGI programming concept.
---	---------------------------------	---

Installation and Configuration

	General Installation Information	Describes the required environments for installing the HTTP server, the prerequisite software, the distribution media, and the general installation process.
	Installation on OS/390 or MVS	Describes the installation procedure on MVS or OS/390 systems.
	Installation on VSE/ESA	Describes the installation procedure on VSE/ESA systems.
	Verifying the Installation	Describes how to verify successful installation and gives troubleshooting information.
	Customizing and Using the HTTP Server	Describes how to initialize, customize and use the HTTP Server.
	Installing NATURAL CGI	Describes the environment and installation procedures to enable Natural CGI processing.

Reference and Maintenance

	Security	
	Programming CGI Requests	Describes the CGI interface modules and how to call them.
	Running CGI Programs under SMARTS	Provides information about running application programs under SMARTS.
	Support and Maintenance	Describes the procedure for reporting problems and applying ZAPS.
	The HTTP Server User Exit	Describes the user exit interface.
	Messages and Codes	Explains messages and codes issued by the HTTP server components of the SMARTS system, including remedial information.
	Sample Members	Lists the supplied sample jobs and tells you what they are used for.

Introduction to the HTTP Server

This chapter tells you how the HTTP server works. It discusses the SMARTS implementation of the CGI interface for C programs; the SMARTS extensions to CGI for 3GL languages and Natural; and the SMARTS conversational CGI program concept.

This chapter covers the following topics:

- The HTTP Server
 - SMARTS Implementation of the Common Gateway Interface (CGI)
 - SMARTS CGI Extensions
 - The Conversational CGI Program Concept
-

The HTTP Server

The HTTP server, based on the HTTP version 1.1 standard, is built on the capabilities of the SMARTS API and the Software AG Com-plete system. It can form the heart of a worldwide web site on the mainframe operating system and provides the vital link between the mainframe resident data and a company intranet or internet web site.

The HTTP server has the following highlights:

- Any form of data including static HTML pages, GIF files, JAVA classes can be delivered from the mainframe host to an HTTP user. Users of browsers on PCs or UNIX systems can access data on the mainframe.
- For text-based data, the HTTP server handles all EBCDIC/ASCII issues using configurable translation tables.
- Common Gateway Interface (CGI) support is fully integrated with the SMARTS API so that C CGI programs that comply with the CGI 'standard' can run.
- Extensions provided by the HTTP server make it possible for Natural, COBOL and PL/1 programs to operate as the target of CGI requests.
- When SAF security checking is active and the HTTP server is running under Com-plete, the HTTP server interfaces fully with three different modes of operation from running all users with the same SAF authorization to a situation where each and every user must provide a user ID and password to the HTTP server before any requests are serviced.
- When running in the Com-plete environment, one or more HTTP servers listening on different ports can be operational within the same SMARTS.
- Each HTTP server may operate with a different security mode.

SMARTS Implementation of the Common Gateway Interface (CGI)

Note:

General information about the Common Gateway Interface (CGI) protocol is available on the Internet. You can use any search engine available on the Internet to find the CGI information.

SMARTS CGI Input Processing

CGI input is processed according to the standard and depends on the HTTP request method used to invoke the CGI program:

- When the 'GET' HTTP request method is used, the input parameters are provided in the QUERY_STRING environment variable and any attempt to access stdin results in an 'end of file' condition being raised.
- When the 'POST' HTTP request method is used, the input parameters are provided as a stdin stream and may be read using any valid SMARTS function to read stdin input. The amount of input to be expected may be determined from the CONTENT_LENGTH environment variable.

Input Translation

All data from the web browser is submitted in standard 7-bit ASCII codes. The HTTP server translates to EBCDIC and sets up headers in standard EBCDIC format as appropriate.

- When the input content type, as specified by the CONTENT-TYPE HTTP header, indicates that the input is type TEXT, the input is translated from ASCII to EBCDIC and is made available to the application program in EBCDIC format.
- When the input content type is TEXT/X-WWW-URLENCODDED, any hex values in the data as specified by %xx where 'xx' is the ASCII hex code for the character being submitted to the CGI request, have the 'xx' translated to the appropriate EBCDIC hex representation of that character. The application may then safely convert this value to a character and be sure that it will be the desired EBCDIC equivalent of that character.

For example, the equals character '=' has an ASCII representation of X'3D' while it has an EBCDIC representation of X'7E'. In the TEXT/X-WWW-URLENCODDED stream, it appears as '%3D'. The HTTP server processing converts this to '%7E' so that the user program sees the EBCDIC representation of the character.

SMARTS CGI Output Processing

CGI output is sent to the stdout stream using any standard C functions normally used to send output to stdout.

Header Processing

The application may provide all, some, or none of the HTTP headers, which provides the greatest flexibility in responding to requests. Two forms of output headers can be supplied:

- nonparsed header (NPH) output; and
- parsed header output.

Nonparsed Header Output

Nonparsed header (NPH) output is supported. It is identified by the string 'HTTP' sent as the first four characters of output. The SMARTS CGI processing routines simply pass all data to the browser directly from the CGI application, so the CGI application can send any header it wishes and any data it likes. The CGI program is then completely responsible for accuracy.

Note:

If a CGI application sends all HTTP headers itself, conversational processing is not supported for the HTTP server.

Parsed Header Output

In most cases, a user CGI program provides a single HTTP header: the CONTENT-TYPE. This is sent as the first string in response to a CGI request and is followed by two CRLF sequences indicating that the data follows (headers are followed by only one CRLF sequence). Optionally, the application may include more than the CONTENT-TYPE header before sending the CRLFCRLF sequence followed by data.

If SMARTS does not detect a CONTENT-TYPE header in the first output from the CGI program, it includes a CONTENT-TYPE header using the default content type for the HTTP server processing the request. Following the CONTENT-TYPE header, the user may submit zero or more HTTP headers.

Alternatively, the LOCATION header may be sent as the first (and only) header. No CONTENT-TYPE header is sent and the LOCATION header is handled according to the HTTP protocol.

Providing No HTTP Headers

When a program sends no HTTP headers at all, which happens if an application was written to stdout in a number of environments, the HTTP server inserts a default CONTENT-TYPE header as specified in the DFLCONT configuration parameter. Software AG recommends that you set this parameter to TEXT/PLAIN to accommodate all programs. Other content types may cause confusing results.

Output Translation Processing

All HTTP headers, whether generated by SMARTS or the CGI program, are expected in EBCDIC and are translated by the HTTP server to ASCII prior to being sent to the CGI requester. This ensures that application programs have readable headers included instead of setting up ASCII data streams using EBCDIC tools and editors.

SMARTS also monitors the data stream for the start of the content itself, which is signified by a CRLFCRLF sequence in the output data stream (headers are followed by only one CRLF sequence). Once the CRLFCRLF sequence is detected, all other data is treated as output content.

SMARTS translates output content from EBCDIC to ASCII only if the output type sent by the CGI program, or defaulted by the HTTP server, is of type TEXT/*. This includes TEXT/PLAIN, TEXT/HTML but also occurs for any other user TEXT/x-application-* type content headers.

Any other content types are sent through unchanged to the requester of the CGI program. This facilitates downloading binary data of any sort.

SMARTS CGI Environment Variables

The following table summarizes the environment variables and their contents set up as a result of a CGI request processed by SMARTS:

Environment Variable	Contains ...
REQUEST_METHOD	a value indicating the HTTP request method used to generate the request. Normally this is GET or POST and indicates where the input data, if any, can be found.
SERVER_PROTOCOL	the version level of the HTTP protocol used to send the request to the server. Normally this has the form HTTP Vv.r where 'v' and 'r' indicate the level of HTTP being used.
SERVER_PORT	the HTTP port number being used to service requests by this HTTP server.
CONTENT_LENGTH	the length of the input data available on the stdin stream for processing CGI requests generated by the POST request method.
CONTENT_TYPE	the type of the input data available on the stdin stream for processing CGI requests generated by the POST request method.
QUERY_STRING	the parameters provided with a content type of TEXT/X-WWW-URLENCODED for CGI requests generated by the GET request method.
PATH_INFO	the piece of the URL used to invoke the CGI program following the CGI program name.
PATH_TRANSLATED	the physical path used to run the CGI program, which may differ on some systems from the URL path. The translated path is always the same as the URL path when running under the HTTP server.
SCRIPT_NAME	the piece of the URL up to the end of the CGI program name. SCRIPT_NAME and PATH_INFO together compose the URL.
REMOTE_USER	the remote user ID provided with the HTTP request.

In addition to the above, all HTTP headers are set up as environment variables by prefixing them with the string 'HTTP_' and translating all '-' or dash characters in the HTTP header name to '_' or underscore characters as per the CGI standard. For example, the header 'LOCATION' may be obtained by requesting 'HTTP_LOCATION' if this header was provided on the request.

Note:

All HTTP environment variable names are uppercase EBCDIC values.

SMARTS CGI Termination Processing

When a CGI program terminates normally, it sends all output data to the stdout stream and returns control to the HTTP server. The HTTP server ensures that all data is sent to the web browser and then closes the connection.

When the HTTP client browser supports persistent sessions, the data is sent to the browser but instead of closing the connection, the HTTP server waits for another request over the same connection.

When a CGI program terminates abnormally either by ABENDING or not returning control to the HTTP server, the web browser may receive some or none of the data sent by the CGI program. In this case, the HTTP server's priority is to clean up the session by closing the connection and freeing all resources associated with that request, thus preventing resource 'leaks' that would otherwise impact the integrity of the HTTP server later in its cycle.

The HTTP server configuration parameter SEND determines the amount of data seen:

- If SEND=IMMEDIATE is set, the client browser sees all data written to stdout to the point where the program ABENDs. Software AG recommends that you set this option in a test environment.
- If SEND=BUFFERED is set, the client browser only sees data if the buffer was filled at least once prior to the ABEND. All data in the buffer at the time of the ABEND is lost. Software AG recommends that you set this option in a production environment for greater performance.

SMARTS CGI Extensions

Now that you understand how a CGI request is generated and processed by a CGI program, this section describes in general terms how languages use the SMARTS CGI extensions. The language-specific chapters provide additional detail.

Standard CGI Operation

CGI is essentially a remote procedure call (RPC) type request driven by an HTML page that results in a request being sent to the HTTP server to run a specific program. The request includes data from the browser filled in by the user of the HTML page that generated the request.

The HTTP server

- recognizes the CGI request;
- makes information from the submitted HTML page and information about the actual request available in the execution environment of the CGI application; and
- gives control to the program identified by the request.

Information is made available

- by using the C stdin stream for certain types of input; and
- by setting up well known environment variables that may be accessed using the getenv function.

By writing to the stdout stream, output is submitted back to the requester with the response.

Non-C CGI Programs

The CGI standard was designed specifically for the C language; however, many installations need to leverage their current skills by providing CGI support in other languages.

Since most languages cannot take advantage of the 'stream' approach for accessing data used by the CGI interface, SMARTS provides extensions to the standard to enable languages other than C to access 'input' from a web CGI request and produce output in response to that input. The SMARTS extensions make it possible to write CGI routines in languages that are as powerful and productive as their C counterparts.

Extensions for Other Languages

Languages such as Natural and COBOL are not suited to interpreting streams of data or parameters provided in strings. Implementation in PL/1 is clumsy while Assembler processing of this data is long-winded and time-consuming from a coding point of view.

Rather than adapt other languages to the C way of doing things, SMARTS provides CGI extensions for accessing data submitted in a CGI request and building the response to that request. The CGI extensions are implemented using a simple call mechanism that allows applications to be written in a more structured mode than normal CGI programs; a mode that suits languages like Natural, COBOL, and PL/1.

Every piece of data submitted as part of a CGI request has a field name associated with it. Where data can be entered in a field, the user enters the field name and the data. As CGI programs are designed to respond to the input from a given HTML page, or at a minimum, HTML pages with certain fields defined, there is no need for search strings.

The SMARTS primary user program request interface is called HAANUPR. This interface will be maintained and upgraded in the future.

For compatibility, previous interface modules continue to be supported so that existing applications can run unchanged, but they will not be enhanced.

These interface modules are documented in detail in the chapter Programming CGI Requests in this manual.

The Conversational CGI Program Concept

Since the foundation of the worldwide web, all CGI programming has been non-conversational or "stateless". This means that when a request is issued, the CGI program processes the request, returns the response to the client, and forgets all about that client.

Some CGI applications store information away about the user's state and retrieve it when the user appears again; however, the state information must then be aged as the user may never come back. It also complicates the program in terms of transactional processing over more than one CGI request, identification of users, and so on.

The HTTP Server Solution

The HTTP server conversational concept avoids these problems by enabling an application program to issue a CONVERSE call in the middle of a processing loop. This sends all data previously output by the CGI program to the client browser.

The HTTP server then suspends the application program until the next HTTP request is issued for that user.

When it is received, the appropriate session is restarted by the HTTP server after the call to CONVERSE with all the data from the client browser available to it as well as any data, switches, or database sessions that the CGI program had in local storage prior to the CONVERSE.

It is possible to set timeouts so that the session context is cancelled if a user doesn't respond within a specific period of time (for example, 30 minutes). When the user does respond, it receives a message to the effect that the conversation doesn't exist.

This mechanism has the following advantages:

- A conversational program can save valuable resources as it is only started once per user and remains active until the conversation ends. Normally, a CGI program must be started and terminated for every CGI request.
- Programs can be structured properly with standard loop and parameter gathering techniques.
- The application program can maintain database or any other connections over the duration of the conversation, avoiding the need to connect and disconnect for each CGI request.
- It gives a CGI application the ability to maintain a transaction over a conversation of two or more HTML pages, if necessary.
- It is a more natural way to program instead of having to collect all the context information which will be required the next time input appears from a given user.

Conversational CGI Application Structure

Any CGI program may converse if the server where the CGI program will run is configured to allow conversational programs. This is controlled by the HTTP server CONV configuration parameter, which must be set to YES to allow conversational programs. Because the CGI program uses the ENABLE-CONVERSE, CONVERSE, and DISABLE-CONVERSE functions of the HAANUPR interface, this facility may be used from any programming language including Natural, COBOL, PL/1, and Assembler.

The following pseudo-code illustrates how such an application is structured. The member HOANCONV on the HTTPVRS.SOURCE dataset is an example of a conversational COBOL CGI program. Comments in the following are enclosed using "/*" to start the comment and "*/" to terminate the comment.

```
BEGIN:
/*
  The following call tells the SMARTS HTTP server that you wish to have a
  conversation with the client browser. If conversations are not supported,
  this call will fail
*/
CALL 'HAANUPR' status 'ENABLE-CONVERSE'
```



```

/*
    In the following logic, the program fields would be filled out with the
    initial values to be presented to the client browser.
*/
Set initial values in output HTML page
/*
    The program will always loop indefinitely until the client at the browser
    indicates that it wants to terminate the conversation, or the program itself
    decides to terminate. The client may indicate this by entering a certain
    value in a field or by pressing a specific HTML button. The program may do
    this based on a transaction being completed.
*/
Do while not end of conversation
/*
    The HTML page is built using multiple calls to the HAANUPR interface to
    output the appropriate HTML to the client. This may be preceded by any
    other logic to get data from a database or build data based on time or
    whatever.
*/
CALL 'HAANUPR' status 'PUT-TEXT' data length
CALL 'HAANUPR' status 'PUT-TEXT' data length
..
/*
    The only requirement in terms of output is that the CGI program must ensure
    that it gets control back by using the appropriate URL on its SUBMIT
    buttons. The easiest way is to use a relative URL like '/cgi/program/'
    where 'program' is the name of the CGI program. The CGI program is then
    redispached then next time the SUBMIT button is pressed. If the wrong URL
    is used in this area, the conversational CGI program is never redispached.
*/
CALL 'HAANUPR' status 'PUT-TEXT' data length

```

The following is the end of the DO WHILE loop:

```

*/
END
/*

```

Once you arrive here, the conversation may be terminated:

```

*/
CALL 'HAANUPR' status 'DISABLE-CONVERSE'
/*
    Write a final 'goodbye' message to the user
*/
CALL 'HAANUPR' status 'PUT-TEXT' data length
END

```

General Installation Information

This chapter describes the required environments for installing the SMARTS HTTP server, the prerequisite software, the distribution media, and the general installation process.

This chapter covers the following topics:

- Supported Operating Systems
 - Supported Environments
 - Installation Media
 - Installation Overview
-

Supported Operating Systems

The SMARTS HTTP server runs in any operating system supported by Com-plete 6.1 (or above) or the SMARTS server environment.

Supported Environments

The SMARTS HTTP server runs in the SMARTS server environment and on Com-plete version 6.1 (or above). Refer to the SMARTS Installation and Operations Manual for information about installing the SMARTS server environment.

SMARTS can interface with any of the following TCP/IP stacks:

- IBM TCP/IP version 3.1 or above.
- IBM TCP/IP version 3.1 with PTF UN93769 and all its pre- and co-requisite PTFs applied.
- IBM TCP/IP version 3.2 with PTF UN99683 and all its pre- and co-requisite PTFs applied. If PTF UQ09354 is applied, PTF UQ11919 must also be applied; otherwise, the system hangs.
- Interlink TCP/IP version 3.1 or above.
- VSE/ESA version 2.2 or above (3Q 1999) using the Connectivity Systems TCP/IP 4 VSE stack.
- MSP/EX from Fujitsu (3Q 1999) using the TISP TCP/IP stack.
- VM/ESA (4Q 1999) using the standard IBM stack available on VM.

The SMARTS HTTP server requires either Com-plete 6.1 (or above) or Natural 3.2 (or above) to support the Natural CGI processing.

The SMARTS Software Developer Kit (SDK) is required to support CGI programs based on C.

Installation Media

For OS/390 or MVS/ESA operating systems, SMARTS can be delivered as a self-extracting executable file. As such, it can be delivered any way that a binary file can be delivered to a user:

- 3.5 inch diskettes as part of the shrink-wrapped package
- using e-mail, both the software itself and updates to the software or fixes

For other operating systems, SMARTS is distributed on tape or cartridge.

Installation Overview

The installation comprises the following steps:

1	If appropriate, execute the self-extracting file to create the required files on a PC hard disk.
2	If appropriate, load the PC files to mainframe datasets as sequential files.
3	Create the mainframe installation datasets as required from the uploaded files, or from the installation tape or cartridge.
4	Run the various installation jobs.
5	Customize to achieve a running system. The customization required is minimal.
6	Verify the installation.

Software AG recommends that you follow the installation procedure step by step. Once the system is running, proper change control ensures that any subsequent changes that cause problems can be backed out smoothly and quickly.

Installation on OS/390 or MVS

Software AG recommends that you keep unmodified copies of all materials distributed or created as part of the installation process. This may assist with problem diagnosis later by providing an untouched sample of any given item.

Note:

Additional installation steps for Natural CGI are documented in the chapter Installing Natural CGI

This document covers the following topics:

- The Installation Tape
 - Creating the Mainframe Datasets
 - Running the Installation Jobs
 - Installing under the SMARTS Server Environment
 - Installing under Com-plete
 - Where Next ?
-

The Installation Tape

Tape Contents

Datasets

The following table lists the product datasets, what the dataset contains, and how it is created. While you are free to rename the datasets, the dataset names used in the table are used consistently throughout the product documentation to ensure clarity.

Distributed Datasets

Dataset	Contains ...
HTPvrs.LOAD	all load modules required by SMARTS HTTP server
HTPvrs.SRCE	all sample source members and macros
HTPvrs.JOBS	all sample JCL required
HTPvrs.INPL	an INPL file for Natural modules and example programs
HTPvrs.UPDW	an INPL update for Natural Web Interface compatibility
HTPvrs.GIFS	GIF files for the Natural Web Interface example demo

Datasets Created during the Installation Process

Name	Dataset containing ...
HTPvrs.USER.SRCE	source members
HTPvrs.USER.LOAD	load modules for all SMARTS environments

Creating the PC Files

Step 1: Copy the installation files to disk

- Copy all of the installation files to a directory on a hard disk where a minimum of 4MB of disk space must be available on a temporary basis in addition to the installation files.

The files are called HTPvrs#n.EXE where 'vrs' is the version, revision, and system maintenance level of SMARTS and 'n' is a sequential number depending on the number of files provided.

All files must be copied from the installation media.

Step 2: Execute each file

- Execute each file provided which will expand to create one or more new files in the same directory.

The files created are listed in the following table along with a description of their contents:

File Name	Contents
\$ALLOC	Sample JCL to allocate the necessary datasets on the mainframe into which the PC datasets will be uploaded. This job is referred to later in this document.
\$TSORECV	Sample JCL to create the actual SMARTS installation datasets from the uploaded datasets. This job is referred to later in this document.
LOAD	HTPvrs.LOAD in off-loaded format.
SOURCE	HTPvrs.SRCE in off-loaded format.
JOBS	HTPvrs.JOBS in off-loaded format.
INPL	HTPvrs.INPL in off-loaded format
UPDW	HTPvrs.UPDW in off-loaded format
GIFS	HTPvrs.GIFS in off-loaded format.

Creating the Mainframe Datasets

Step 1: Allocate the Datasets

- Allocate a dataset with the following DCB information for each off-loaded file that now exists on the PC:

RECFM=FB
 LRECL=80
 BLKSIZE=3120

Software AG recommends that you

- name each dataset you allocate based on its HTPvrs.* name with the suffix .SEQ. For example, the dataset for the PC LOAD file would be HTPvrs.LOAD.SEQ.
- allocate the datasets in blocks. You can determine the number of blocks required by dividing the size of the PC file in bytes by the blocksize 3120 and adding 1. For example, if the PC file is 3,480 bytes, allocate 2 blocks.

You may allocate the datasets using either

- TSO; or
- the \$ALLOC file, which may be uploaded to a source dataset on the mainframe as a text file (using ASCII/EBCDIC translation) and modified to suit the installation requirements and to reflect the correct space allocation required for each dataset.

Step 2: Upload the Data

- Once you have allocated the sequential datasets on the mainframe, use a binary transfer to upload the binary files to their equivalent mainframe dataset.

IND\$FILE and standard FTP implementations all offer the binary transfer capability.

The following table pairs the PC file with the mainframe dataset to which it should be loaded:

PC File	Mainframe Dataset
LOAD	HTPvrs.LOAD.SEQ
SRCE	HTPvrs.SRCE.SEQ
JOBS	HTPvrs.JOBS.SEQ
INPL	HTPvrs.INPL.SEQ
UPDW	HTPvrs.UPDW.SEQ
GIFS	HTPvrs.GIFS.SEQ

Step 3: Create the SMARTS Datasets

- Once you have uploaded the data to the mainframe, the actual mainframe installation datasets must be created. Because the sequential datasets uploaded from the PC are actually the result of a TSO TRANSMIT command, the datasets must be recreated using a TSO RECEIVE command.

You may either

- upload the \$TSORECV file on the PC to a source dataset on the mainframe as a text file (using ASCII/EBCDIC translation) and use it as a sample to issue the appropriate commands in batch; or

- RECEIVE the datasets individually using TSO commands.

Software AG recommends that RECEIVE be allowed to allocate the space required by the datasets as it can determine what is required from internal information in the sequential file itself.

Once the SMARTS datasets have been created, delete the sequential datasets created for the data uploaded from the PC (that is, the datasets with the .SEQ suffix). These can be created again from the PC files, if necessary.

Running the Installation Jobs

The following procedure installs the SMARTS HTTP server product.

Step 1: Allocate and Initialize User PDS Datasets

- To create and initialize the datasets required for the SMARTS HTTP server, modify the sample job in member \$JCLALLO on the HTPvrs.SRCE dataset to suit your installation's environment, and run it to create the appropriate datasets.

This job also copies all modifiable members from the HTPvrs.SRCE dataset to the newly created HTPvrs.USER.SRCE dataset in order to retain all HTPvrs.SRCE members as delivered.

Note:

To ensure that the \$JCLALLO member remains as delivered on the HTPvrs.SRCE dataset for future reference, Software AG recommends that you modify and submit the job from the editor without saving it. Once the job has completed successfully, the job may be saved in the HTPvrs.USER.SRCE dataset.

Step 2: Install the HTTP Server HLL Interface Modules

- A number of high-level language interface modules must be linked in order to execute the HTTP server.

Modify and run the sample JCL member HJENLINK in the HTPvrs.JOBS dataset to link the modules in the appropriate way to the HTPvrs.USER.LOAD dataset.

Ensure that the job finishes with condition code "0".

Step 3: Install the Natural INPL File

-

Note:

This step applies only if you are running with Natural.

Install the INPL file delivered with the SMARTS HTTP server creating a Natural library called HTPvrs.

Refer to the chapter Installing Natural CGI for information about installing Natural CGI support.

Step 4: Install the Natural INPL Update File

-

Note:

This step applies only if you are running with Natural 3.1 or above and the Natural Web Interface.

Install the INPL update file HTPvrs.UPDW delivered with SMARTS to update the SYSWEB library.

Refer to the chapter Installing Natural CGI for information about installing Natural Web Interface support.

Step 5: Customize the HTTP Server

- The sample configuration member HAANCONF was copied into the HTPvrs.USER.SRCE dataset during installation. Software AG recommends that you use the member as delivered for the installation verification routines unless the default port number (8080) is inappropriate.

If the HTTP server is to be used with Natural, it may be necessary to change the NATTHRD and NATLIB parameters.

If it is necessary to change the port number or any other parameter (the default port is 8080 and is already set in the supplied default HTTP configuration):

1. Modify the parameters in HAANCONF.
2. Recompile HAANCONF using the sample HJBNA CNF JCL in HTPvrs.JOBS.
3. Assemble HAANCONF to the HTPvrs.USER.LOAD library available to the server at startup.

Refer to the HTTP Server Use and Customization chapter later in this manual for details about this process and the parameters that can be specified.

Step 6: Customize the SMARTS Environment

- The SMARTS environment configuration member PXANCONF must include the communication driver interface (CDI) protocol definition for cgistdio.

Following is a sample of the CDI_DRIVER parameter specification:

```
CDI_DRIVER=( 'cgistdio,HAANPCGI' )
```

See the SMARTS Installation and Operations Manual for more information about this step.

Installing under the SMARTS Server Environment

Step 1: Modify the SMARTS Server Start-up Procedure or Job

- Include the following datasets in the COMPLIB dataset concatenation in the order shown:

```
// DD DISP=SHR,DSN=HTPvrs.USER.LOAD
// DD DISP=SHR,DSN=HTPvrs.LOAD
```


You may optionally add the following DD statement in the JCL to direct output to a specific job class or dataset:

HTPTRCE	the HTTP server trace output when HTTP server tracing is active
---------	---

Step 2: Modify the SMARTS Server Start-up Parameters

- The member HJENPARM on the HTPvrs.JOBS dataset provides a sample set of the parameters required by the SMARTS server to
 - start the HTTP server; and
 - define the various interface extensions as RESIDENTPAGE.

Add the parameters in HJENPARM into your standard set.

At least one thread with 404 kilobytes defined below the 16 megabyte line and 1 megabyte above the line is required to run the installation verification programs. The values are not absolute and may be reduced depending on the server usage and the language environment configuration.

Installing under Com-plete

Note:

The steps in this section apply only if you are running the SMARTS HTTP server under Com-plete.

The procedure described in this section installs the SMARTS HTTP server under Com-plete.

Step 1: Modify the Com-plete Start-up Procedure or Job

- Include the datasets in the COMPLIB dataset concatenation as for the SMARTS server environment.

Step 2: Modify the Com-plete Start-up Parameters

- Add the parameters from the sample member HJENPARM to the standard set as for the SMARTS server environment.

Step 3: Catalog PAENSTRT

- Use the ULIB utility of Com-plete to catalog the PAENSTRT program with an initial thread size of 400 kilobytes.

Step 4: Install the LE in Com-plete

- To test the C, PL/I, and COBOL programs delivered with SMARTS, you must be able to run language environment (LE)-enabled programs in the Com-plete system.

Refer to the Com-plete documentation for information.

Step 5: Verify the Installation

- See the chapter Verifying the Installation.

Execute the steps for the SMARTS server environment against Com-plete to ensure that SMARTS is running successfully under Com-plete.

In addition to running the programs using the HTTP server, it should be possible to execute the programs from the command line of a Com-plete session. The output appears in the stdout file.

Step 6: Restart Com-plete

- Near the end of initialization processing, messages are issued to the console indicating that the HTTP server has been started.

If the server does not start successfully, check for error messages and verify the installation steps again.

Where Next ?

Continue to the chapter Verifying the Installation for information about verifying the SMARTS HTTP server installation and troubleshooting.

Then familiarize yourself with the customization and configuration options available in the product. Following the customization sections are a number of sections detailing specific functionality and how to implement this functionality in the SMARTS environment.

For specific information about the programming interfaces and how to use them, refer to the chapter Programming CGI Requests later in this manual.

Installation on VSE/ESA

Software AG recommends that you keep unmodified copies of all materials distributed or created as part of the installation process. This may assist with problem diagnosis later by providing an untouched sample of any given item.

This document covers the following topics:

- The Installation Tape
 - Installing under the SMARTS Server Environment
 - Installing under Com-plete
 - Where Next ?
-

The Installation Tape

Tape Contents

Datasets

The following table lists the product datasets, what the dataset contains, and how it is created. While you are free to rename the datasets, the dataset names used in the table are used consistently throughout the product documentation to ensure clarity.

Note:

Software AG recommends that you use the default values to ensure that the HTTP server will install and execute without the need for user intervention.

Distributed Datasets

Dataset	Contains ...
SAGLIB.HTTPvrs	all components required by the SMARTS HTTP server
HTTPvrs.INPL	an INPL file for Natural modules and example programs
HTTPvrs.UPDW	an INPL update for Natural Web Interface compatibility
HTTPvrs.GIFS	GIF files for the Natural Web Interface example demo

Libraries and Sublibraries Created during the Installation Process

Name	Dataset containing ...
SAGLIB.HTTPvrs	all components required by the SMARTS HTTP server

Running the Installation Jobs

The following procedure installs the SMARTS HTTP server product.

Step 1: Copy the HTTP Server Components to Disk

- Use the LIBR Restore function to copy the HTTP server components from the tape into the library or sublibrary of your choice.

Software AG recommends that you use the library SAGLIB and the sublibrary HTPvrs.

Step 2: Install the Natural INPL File

-

Note:

This step applies only if you are running with Natural.

Install the INPL file delivered with the SMARTS HTTP server creating a Natural library called HTPvrs.

Refer to the chapter Installing Natural CGI for information about installing Natural CGI support.

Step 3: Install the Natural INPL Update File

-

Note:

This step applies only if you are running with Natural 3.1 or above and the Natural Web Interface.

Install the INPL update file HTPvrs.UPDW delivered with SMARTS to update the SYSWEB library.

Refer to the chapter Installing Natural CGI for information about installing Natural Web Interface support.

Step 4: Customize the HTTP Server

- The sample configuration member HAANCONF.J was copied into the HTPvrs sublibrary during installation. Software AG recommends that you use this member as delivered for the installation verification routines unless the default port number (8080) is inappropriate.

If the HTTP server is to be used with Natural, it may be necessary to change the NATTHRD and NATLIB parameters.

If it is necessary to change the port number or any other parameter (the default port is 8080 and is already set in the supplied default HTTP configuration):

1. Modify the parameters in HAANCONF.A.
2. Assemble HAANCONF using the sample HJBNA CNF.J JECL in the HTPvrs sublibrary.
3. Generate the resulting phase into the HTPvrs sublibrary available to the server at startup.

Refer to the HTTP Server Use and Customization chapter later in this manual for details about this process and the parameters that can be specified.

Step 5: Customize the SMARTS Environment

- The SMARTS environment configuration member PXANCONF must include the communication driver interface (CDI) protocol definition for cgistdio.

Following is a sample of the CDI_DRIVER parameter specification:

```
CDI_DRIVER=( 'cgistdio,HAANPCGI' )
```

See the SMARTS Installation and Operations Manual for more information about this step.

Installing under the SMARTS Server Environment

Step 1: Modify the SMARTS Server Start-up Job

- Add the HTTP server library and sublibrary to the LIBDEF concatenation.

Step 2: Modify the SMARTS Server Start-up Parameters

- The member HJENPARM.P in the HTPvrs sublibrary provides a sample set of parameters required by the SMARTS server environment to
 - start the HTTP server parameters; and
 - define the various extensions as RESIDENTPAGE.

Add the parameters from the sample member HJENPARM.P to SYSPARMS section of your SMARTS server start-up JECL.

At least one thread with 404 kilobytes defined below the line and 1 megabyte above the line is required to run the installation verification programs. The values are not absolute and may be reduced depending on server usage and the language environment configuration.

Installing under Com-plete

Note:

The steps in this section apply only if you are running the SMARTS HTTP server under Com-plete.

The procedure described in this section installs the SMARTS HTTP server under Com-plete.

Step 1: Modify the Com-plete Start-up Job

- Add the HTTP server library and sublibrary to the LIBDEF concatenation as for the SMARTS server environment.

Step 2: Modify the Com-plete Start-up Parameters

- Add the parameters from the sample member HJENPARM.P to the SYSPARMS section of your start-up JECL as for the SMARTS server environment.

Step 3: Catalog PAENSTRT

- Use the ULIB utility of Com-plete to catalog the PAENSTRT program with an initial thread size of 400 kilobytes.

Step 4: Install the LE in Com-plete

- To test the C, PL/I, and COBOL programs delivered with SMARTS, you must be able to run language environment (LE)-enabled programs in the Com-plete system.

Refer to the Com-plete documentation for information.

Step 5: Verify the Installation

- See the chapter Verifying the Installation.

Execute the steps for the SMARTS server environment against Com-plete to ensure that SMARTS is running successfully under Com-plete.

In addition to running the programs using the HTTP server, it should be possible to execute the programs from the command line of a Com-plete session. The output from these programs is written to SYSLST.

Step 6: Restart Com-plete

- Near the end of initialization processing, messages are issued to the console indicating that the HTTP server has been started.

If the server does not start successfully, check for error messages and verify the installation steps again.

Where Next ?

Continue with Verifying the Installation to ensure the installation was correct, and for any troubleshooting information.

Then familiarize yourself with the customization and configuration options available in the product. Following the customization sections are a number of sections detailing specific functionality and how to implement this functionality in the SMARTS environment.

For specific information about the programming interfaces and how to use them, refer to Programming CGI Requests.

Verifying the Installation

Use online commands to verify the SMARTS HTTP server installation.

This chapter covers the following topics:

- Verify Operation of the Servers
 - Verify the SMARTS HTTP Server Installation
 - Troubleshooting
-

Verify Operation of the Servers

The sample programs and HTML files described in this section are used to verify the operation of the HTTP server.

Sample Programs

The following table lists the sample programs used for each verification stage and the output they produce. All programs are also provided in source format.

Program	Sample . . .	Output
HCANSAMP	C program that takes a content value as CGI input and sends it back to the user.	The value sent as the content value.
HOANSAMP	COBOL CGI program that shows the uses of HAANUPR.	The value sent as the content value.
HOANCONV	COBOL CGI program that makes use of conversational mode.	The value sent as the content value and the previous value.
HPANSAMP	PL/I CGI program that shows the uses of HAANUPR.	The value sent as the content value.

Sample HTML Files

The following table lists the sample HTML documents used during verification processing where:

ip-addr	is the IP address of the TCP/IP subsystem where the HTTP server is running
port	is the port number specified in your HTTP server configuration

Note:

You must change the URL to reflect your version of the dataset name. For example, if HTTPvrs.TEST.USER.SRCE is your name for the dataset referred to in this document as HTTPvrs.USER.SRCE, the first URL in the following table would be:
<http://ip-addr:port/httpvrs/test/user/srce/PXANCONF>

http://ip-addr:port/	Description
httpvrs/user/src/PXANCONF	a basic test of the HTML functions; displays the PXANCONF configuration file in text format on a WWW browser.
httpvrs/src/hhancget.htm	sample HTML form that drives the sample C CGI program HCANSAMP for HTTP GET processing.
httpvrs/src/hhancobt.htm	sample HTML form that drives the sample COBOL CGI program HOANSAMP for HTTP GET processing.
httpvrs/src/hhanplt.htm	sample HTML form that drives the sample PL/1 CGI program HPANSAMP for HTTP GET processing.
httpvrs/src/hhannatt.htm	sample HTML form that drives the sample Natural CGI program HNANSAMP for HTTP GET processing. Note: Natural CGI processing capability must be installed before this HTML can be run.
cgi/hoanconv	runs the sample COBOL conversation CGI program HOANCONV. HOANCONV ‘ages’ the last two pieces of data as it is entered, illustrating how user context can be maintained over multiple HTML outputs. Enter ‘stop’ to terminate the conversation.
cgi/<program>	runs <program> which may be any sample program from the previous section Sample Programs; displays program output on a WWW browser.

Verify the SMARTS HTTP Server Installation

The HTTP server installation is verified by running the HTTP server and by accepting and processing HTTP requests from a standard browser.

Prepare the Sample Programs

Compile the C sample HCANSAMP and link it using the SMARTS headers and stubs. See the SMARTS Installation and Operations Manual for details of example jobs.

Refer to the chapter Installing Natural CGI before attempting to run the Natural CGI test programs.

Start the HTTP Server

Start up the SMARTS server environment and ensure the HTTP server starts correctly.

To verify that the HTTP server is operating correctly, run each of the tests described in Sample HTML Files.

Troubleshooting

The HTTP Server Initialization Fails

If the SMARTS environment initialization fails, the HTTP server initialization will also fail.

If the SMARTS environment initialization fails, its configuration is probably invalid. See the SMARTS environment installation and verification information in the SMARTS Installation and Operations Manual.

The SMARTS Environment Initializes, but the HTTP Server Initialization Fails

The HTTP server configuration is probably invalid.

- Check for messages during the HTTP server initialization process that may identify the problem.
- Check in particular for sockets errors that occur if the specified port is already in use by another application in your system.

All SMARTS Components Initialize, but Access Attempts Fail

If SMARTS components issue messages about successful completion of tasks, the problem is probably in the request being issued or the port number being used. Ensure that

- the IP address or DNS name used to identify the target host actually identifies the TCP/IP stack with which SMARTS is communicating. More than one IP address is possible on a mainframe system: if you use the wrong one, your request will time out or be rejected.
- you can connect to the IP addresses by pinging that node. If the ping fails, you have no physical connection to that host.
- the port specified is the one the HTTP server is configured to work with. The default is port 8080.

Customizing and Using the HTTP Server

This chapter covers the following topics:

- Initializing the HTTP Server
 - Termination
 - Operator Commands
 - Configuration
 - HTTP Server Parameters
 - Content Processing
 - Configurable Tables
 - Default URL Processing
 - Resource Usage
 - Pooled Server Processing
 - Conversational Processing
-

Initializing the HTTP Server

The HTTP server is normally initialized by specifying a **SERVER** statement:

```
SERVER=(HTTP,HAENSERV,Configuration=<config>)
```

-where

HTTP	is the name of the SERVER. The name of the server may be any name valid for the SERVER statement.
HAENSERV	is the name of the HTTP server module. The HTTP server module must always be specified as HAENSERV.
<config>	is the name of the HTTP server configuration to be used. The configuration statement may specify any valid load module name. The load module must have been generated as described in this section; otherwise, the results are unpredictable. The default configuration name used is HAANCONF.

If inactive for any reason, the HTTP server may also be started using the operator command

```
SERV,INIT,HTTP,HAENSERV,Configuration=<config>
```

-where the meaning of the parameters is the same as previously outlined for the **SERVER** statement.

Notes:

1. 1. It is possible to have more than one HTTP server active within the same SMARTS server address space at the same time. Each server requires a different server name and a different configuration to ensure that it does not attempt to use the same port as one of the other servers.
2. 2. The SMARTS environment must be active before any the HTTP server can be activated; otherwise, the HTTP server initialization fails. This can be achieved by placing any of the HTTP server SERVER statements after the SERVER statement for the SMARTS environment in the SMARTS server sysparms input file. When the servers are started using operator commands, the SMARTS environment must be started first.
3. . The QUIESCE command should normally be issued first to give the server time to stop accepting new requests and finish processing any old requests.
4. . The first time the command is issued, the listening program attached when the server was initialized is canceled if it is still active. In this case, the terminate must be requested again as the HTTP server cannot terminate properly until its associated listening task has terminated.
5. . If the SMARTS environment is terminated, all the HTTP server listening tasks are automatically canceled.
6. . It is not possible to terminate the SMARTS environment unless all the HTTP servers are first terminated.

Termination

The HTTP server is terminated automatically when the SMARTS server address space is terminated. However, it is also possible to cycle the server without bringing the address space down by issuing the operator commands

```
SERV,HTTP,QUIESCE
SERV,TERM,HTTP
```

-where HTTP is the name given to the server at startup.

Notes:

1. . The QUIESCE command should normally be issued first to give the server time to stop accepting new requests and finish processing any old requests.
2. . The first time the command is issued, the listening program attached when the server was initialized is canceled if it is still active. In this case, the terminate must be requested again as the HTTP server cannot terminate properly until its associated listening task has terminated.
3. . If the SMARTS environment is terminated, all the HTTP server listening tasks are automatically canceled.
4. . It is not possible to terminate the SMARTS environment unless all the HTTP servers are first terminated.

Operator Commands

In addition to the SMARTS server commands to initialize and terminate the HTTP server, the following operator commands may be issued to the HTTP server by issuing the SMARTS server operator command

`SERV,HTTP,<command>`

-where

HTTP	is the name with which the HTTP server was started.
<command>	is one of the commands in the following table:

Command	Function
CLEARPOOL	Terminates all active HTTP pooled servers. If no pooled servers are in operation, this command has no effect.
QUIESCE	Causes the HTTP server to stop accepting new requests while enabling existing requests to continue normally. Software AG recommends issuing this command to the HTTP server before attempting to terminate it.
TERM	Terminates the HTTP server as gracefully as possible.
FORCE	Forcibly terminates the HTTP server. This command should only be used in emergencies as it may cause integrity problems.

Configuration

The HTTP server is configured by building a load module with the HMANCONF macro delivered with SMARTS.

Under the SMARTS server, the load module is identified on the SERVER statement or on the operator command used to start the server.

The load module is specified using the 'Configuration=' parameter and defaults to HAANCONF.

Any attempt to use a module that was not generated according to the instructions in this section will cause unpredictable results.

Sample HAANCONF Member

The HTPvrs.USER.SRCE statement contains a sample HAANCONF member. This may be copied and/or modified to produce a number of different configuration options, if required. A configuration option can then be selected by an operator when SMARTS is started.

Assembling the Configuration Member

Once the configuration source member has been created or modified, it must be assembled to produce the associated load module for use by the system.

It must be linked into a load library in the COMPLIB concatenation of the SMARTS server environment start-up procedure.

Software AG recommends that the HTPvrs.USER.LOAD dataset contain all configuration modules.

The member HJBNA CNF on the HTPvrs.USER.SRCE dataset contains sample JCL to compile and link the delivered HAANCONF member.

HTTP Server Parameters

The following parameters may be specified on the HMANCONF macro:

CGIPARM

Parameter	Use	Possible Values	Default
CGIPARM	A parameter string to be passed to every CGI program that is started.	1-256 byte parameter string	none

The string is passed in standard OS/390 or MVS/ESA format where register 1 points to a pointer that points to a half word length followed by the data.

CGIPARM is used to pass parameters or set runtime options for language environment-enabled programs.

CONTBUFL

Parameter	Use	Possible Values	Default
CONTBUFL	The size of a buffer allocated by the HTTP request processing program to hold any HTTP content submitted with an HTTP request.	1-32000	1024

If the content size exceeds the value set in this parameter, the request is rejected.

Changing this size affects the amount of local storage that the request processing program needs to run.

CONV

Parameter	Use	Possible Values	Default
CONV	Indicates whether the server supports conversational users.	NO YES	NO

If NO, any program request to establish a conversation is rejected with an appropriate return and reason code.

DEFACEE

Note:

This parameter is only used in a SMARTS server environment.

Parameter	Use	Possible Values	Default
DEFACEE	Indicates whether the server will build a default ACEE for the user specified in HTTPUSER during initialization and startup.	NO YES	NO

Value	Description
YES	A default ACEE is built and associated with any user that does not log on to the system.
NO	Any user that does not log on to the system has the authority associated with the address space.

DFLTCONT

Parameter	Use	Possible Values	Default
DFLTCONT	The default content type to be assigned to a URL if the content type cannot be determined through the dataset name or member type processing mechanism.	HTTP content type header	APPLICATION/OCTET-STREAM

The web browser uses the content type header to determine what to do with data. For example, for content type TEXT/HTML, the browser interprets the data as an HTML page. For content type IMAGE/GIF, the browser attempts to interpret the data as a GIF file.

The default value causes the web browser to download as binary data any URLs for which the content cannot be explicitly determined; that is, such URLs are downloaded as is without any translation.

DFLTURL

Parameter	Use	Possible Values	Default
DFLTURL	The default URL to be returned to a user request connecting to the HTTP server with no URL information.	default URL for your installation	none

This circumstance occurs when the following URL is requested from a browser:

http://your.ip.address:port

Refer to the subsection later in this section relating to the specification of a default URL.

HTTPUSER

Note:

This parameter is only used in a SMARTS server environment.

Parameter	Use	Possible Values	Default
HTTPUSER	The user ID with which each HTTP request is identified if the HTTP request does not contain an authorization header.	1-8 character user ID	HTTPUSER

This is effectively the default user ID assigned initially to all requests.

HTTPLIST

Note:

This parameter is only used in a SMARTS server environment.

Parameter	Use	Possible Values	Default
HTTPLIST	The user ID assigned to the HTTP task that listens for requests.	1-8 character user ID	HTTPLIST

HTTPHCD

Note:

This parameter is only used in a SMARTS server environment.

Parameter	Use	Possible Values	Default
HTTPHCD	The hardcopy device name associated with any HTTP users.	1-8 character name	HTTPHCD

This output device name is used when the program attempts to write output to a SYSOUT/SYSLST type device. For example, when LE/370 is active and an abend occurs, LE/370 writes a dump to this hardcopy device name.

Output written to this device may be viewed and/or deleted using USPOOL or printed.

LOGON

Note:

This parameter is only used in a SMARTS server environment.

Parameter	Use	Possible Values	Default
LOGON	Determines the level of security that the HTTP server will enforce.	ALLOWED DISALLOWED REQUIRED	ALLOWED

The parameter must be used in association with the SMARTS server SECSYS configuration parameter.

Refer to the chapter on Security for more information about this parameter.

MSGCASE

Parameter	Use	Possible Values	Default
MSGCASE	The case in which messages are to be issued.	MIXED UPPER	MIXED

NATLIB

Parameter	Use	Possible Values	Default
NATLIB	Name of the default library where Natural CGI requests to the HTTP server are directed.	1-8 character name	NATCGI

If no library is provided on the CGI request, the program specified on the Natural CGI request to this server must be cataloged in this library.

NATPARAM

Parameter	Use	Possible Values	Default
NATPARAM	Parameter string to be provided as override parameters to Natural for all Natural CGI requests.	1-256 byte parameter string	none

Note:

The contents of this field are not validated in any way and may cause problems if invalid. Any string should be thoroughly tested before being set in this parameter.

The 'STACK' override parameter is ignored if specified in this string.

NATTHRD

Parameter	Use	Possible Values	Default
NATTHRD	Under the SMARTS server environment, the name of the thread-resident portion of Natural as created for the SMARTS server environment TP monitor.	1-8 character name	NATCOM

See Installing Natural CGI.

PORT

Parameter	Use	Possible Values	Default
PORT	The sockets port on which the HTTP server server should listen for incoming HTTP requests.	1-32000	80

RECVBUFL

Parameter	Use	Possible Values	Default
RECVBUFL	The length of the buffer used for receiving input data from the network.	1-32000	4096

When conversations are being supported, this buffer must be set to the highest incoming data size expected for one incoming conversational HTTP request; otherwise, data may be lost.

When conversations are not being supported, this buffer is reused to read the entire incoming data stream.

Changing this size affects the amount of local storage the request processing program needs to run.

SEND

Parameter	Use	Possible Values	Default
SEND	Determines how HTTP sends data in response to a request.	IMMEDIATE BUFFERED	IMMEDIATE

Value	Description
IMMEDIATE	Sends data as soon as it is available. Although it is more resource intensive, this option is useful where requests are failing: the web user receives whatever data has been created to the point of the failure.
BUFFERED	Buffers all data in the buffer created by the SENDBUFL parameter. If failures occur, the end user may see no response data at all as the data is being buffered; however, this is less resource and Entire Net-Work intensive. Because the BUFFERED setting greatly increases the performance of the server, Software AG recommends that use it in a production environment.

Note:

The SENDBUFL parameter must still be specified with SEND=IMMEDIATE as the data must be copied and translated in some instances prior to being sent.

SENDBUFL

Parameter	Use	Possible Values	Default
SENDBUFL	The length of the buffer used for sending output data to the network.	1-32000	4096

Changing this size affects the amount of local storage the request processing program needs in order to run.

SERVNAME

Parameter	Use	Possible Values	Default
SERVNAME	Name to identify the system.	1-8 character name	none

This name is included in all messages (except some start-up and termination messages) issued to the operator during the execution of the HTTP server.

The name may be used in the future by the HTTP server system to uniquely identify itself within a machine.

SERVPOOL

Parameter	Use	Possible Values	Default
SERVPOOL	Indicates whether the HTTP server maintains a pool of previously started HTTP servers.	NO YES	NO

Value	Description
NO	Pooled servers are not maintained and a new server is started for each HTTP request.
YES	The HTTP server reuses previously started servers, which can significantly enhance performance.

TRACE

Parameter	Use	Possible Values	Default
TRACE	Turns on tracing to the DD statement identified by the TRACEDD keyword.	HEADER DATA	none

One or both options may be specified. The options must be specified in parentheses. For example, the following turns both traces on:

TRACE=(HEADER,DATA)

Value	Description
HEADER	Dumps all HTTP headers and their associated data to the DD once they have all been read and processed.
DATA	Traces all HTTP input data as it is read and all data being sent in response to the request prior to it being sent. The data is printed in both character and hex formats. The hex represents what is actually sent in ASCII. The character output is translated to EBCDIC so that it can be read.

TRACEDD

Parameter	Use	Possible Values	Default
TRACEDD	Name of the DD to which all HTTP trace data is written.	1-8 character name	HTPTRCE

If the DD name is not specified in the SMARTS start-up procedure, it is automatically allocated as a SYSOUT dataset.

URLPBUFL

Parameter	Use	Possible Values	Default
URLPBUFL	Length of the buffer used for holding parameters passed on any given URL request (that is, any data following ? in the URL).	1-32000	256

Depending on the nature of the requests in use, the length could be increased or decreased; however, if there is insufficient space in this buffer, the request is rejected.

Changing this size affects the amount of local storage the request processing program needs in order to run.

Content Processing

One of the most important pieces of information that the HTTP server provides to the web browser in response to a HTTP request is the 'content type'. This is the HTTP header that the browser uses to interpret the data sent in response to the request. The HTTP server has a number of ways to determine what the content type is.

Member Type Processing

When a member type is specified on a URL request, even though the mainframe operating system has no concept of types as such, the HTTP server uses this type to look up a user configuration table HAANTYPT to determine whether it can identify the content type of the URL.

The source for the table HAANTYPT is delivered in HTPvrs.SRCE and copied during installation to the HTPvrs.USER.SRCE datasets for modification by the user. The table is built using the HMANTYPT macro which has two parameters:

TYPE	1-8 character member type.
CONTENT	The content type to be returned to the browser when a member of this type is requested on a URL. The HTTP server does not validate content types: any type may be specified and new types are constantly being created. The sample HAANTYPT member contains examples of commonly used content types.

Note:

The values specified for the above parameters are case-sensitive; for example, HTML and html are different 'TYPE's.

Once modified, the HAANTYPT table must be reassembled into a load library in the SMARTS server environment COMPLIB concatenation.

Software AG recommends that you use the HTPvrs.USER.LOAD dataset for this. Member HJBNTYPT in the HTPvrs.USER.SRCE dataset contains sample JCL to assemble and link the HAANTYPT table.

If a match cannot be found for the type specified on the URL, or no type is specified on the URL, the HTTP server attempts to determine the content type from the last level of the URL provided.

Dataset Name Processing

The HTTP server breaks down the URL provided by the user into

- a dataset name component; and
- optionally a member and member type component

```
/this/is/a/pds/member.type is THIS.IS.A.PDS(MEMBER)
```

-or

```
/this/is/a/seq/file/ is THIS.IS.A.SEQFILE
```

On OS/390 or MVS/ESA systems, the HTTP server assumes that the last level of the resulting dataset name indicates the sort of data contained in a given dataset. In the above examples, 'PDS' and 'FILE' are the last levels.

Once the last level is determined, the HTTP server checks a second user-configurable table HAANDSNT to determine if the last level can be used to map the URL to a content type. The HAANDSNT member is provided on the HTPvrs.SRCE dataset and copied by the installation to the HTPvrs.USER.SRCE for modification. HAANDSNT is built using the HMANDSNT macro which has the following parameters:

LASTDSNL	The last level of the dataset name resulting from the interpretation of the URL.
CONTENT	The content type to be returned to the browser when a member from this dataset (or the dataset itself, if sequential) is requested on a URL. The HTTP server does not validate these content types: any type may be specified and new types are being created daily. The sample HAANDSNT member contains examples of commonly used content types.

Note:

The values specified for the above parameters are case-sensitive; for example, HTML and html are different 'LASTLEVL's. At present, all LASTLEVL specifications should be in uppercase as there is no support for lowercase dataset names.

Once modified, the HAANDSNT table must be reassembled into a load library in the SMARTS server COMPLIB concatenation.

Software AG recommends that you use the HTPvrs.USER.LOAD dataset for this. Member HJBNDST in the HTPvrs.USER.SRCE dataset contains sample JCL to assemble and link the HAANDSNT table.

If a match cannot be found in the HAANDSNT table, the HTTP server uses the default as specified by the DFLTCONT configuration parameter.

CGI Request Output Processing

Any CGI program, whether it is written in Natural, C, COBOL, or PL/1, may write a CONTENT-TYPE header to the output stream to indicate the content it is going to send. If the HTTP server detects that no CONTENT-TYPE header is sent by the CGI program, it sends a CONTENT-TYPE header using the default content type for the system as specified by the DFLTCONT configuration parameter.

Configurable Tables

A number of translation tables are supplied with the HTTP server in source format.

Do not modify these tables, as incorrect modification may cause server problems for the server.

If you find errors in these tables, contact Software AG so that corrections can be made generally available.

HAANEUTT

The HAANEUTT table translates from EBCDIC to uppercase EBCDIC.

HAANIPTT

The HAANIPTT table translates from ASCII to EBCDIC.

HAANIUTT

The HAANIUTT table translates from ASCII to uppercase EBCDIC.

HAANOPTT

The HAANOPTT table translates from EBCDIC to ASCII.

HAANTOTT

The HAANTOTT table translates trace character output to ensure valid output data.

Default URL Processing

Care must be taken with the default URL specified for the HTTP server. If a file containing an HTML page is set as the default, this file must not contain any relative URLs as any attempt to reference these relative URLs from a page accessed as the default URL will fail.

The reason for this is that the browser sends a request for the dataset named '/' which causes the HTTP server to use the default URL specified in the configuration parameters. If this is a dataset, it will have the form

```
/a/b/c/member.type
```

-and 'member' will be returned to the caller.

Any relative URLs in that member are relative to /a/b/c/; however, the browser does not know this and assumes that they are relative to '/', which is how it originally found it.

The browser translates the URL './graphics/my.gif' to be '/graphics/my.gif' when, in fact, the URL should be '/a/b/c/graphics/my.gif'.

As the HTTP server knows nothing about a dataset called simply 'graphics' with a member of 'my', the request fails.

There are two ways to avoid this problem:

- In the default URL, specify absolute URLs only. This must only be done for one HTML file and as such should not be a major problem to maintain.
- Specify a CGI program as the default URL which either directs the web user to the correct URL or rejects the request depending on how secure the server should be.

Resource Usage

The HTTP server itself uses a combination of local storage and global storage acquired explicitly by direct requests for the storage and implicitly by using the SMARTS API requests.

The following sections outline the use of HTTP server storage.

Global Storage

The following storage areas used by the HTTP server are allocated in global storage above the 16 megabyte line in all environments:

Storage Area	Size in Bytes
HTTP main control block (HMCB)	2048
Server storage stack	4096
Storage for pool server processing	<no of pooled servers>*64
Storage for conversational users	<no of conversations>*64
Storage for data pipe between processes. This is relatively short term as it is allocated, written, read, and freed.	<max of RECVBUFL> *<concurrent pipes>

HAANLIST Storage

The HAANLIST program listens on the specified port for HTTP requests and is fully reentrant. The following storage areas are allocated for its processing needs:

Storage Area	Size in Bytes
Working storage	256
Storage required for 2 sockets	see the SMARTS environment estimates
Various working storage areas	1024

HAANRQST Storage

This program actually processes the HTTP request and is fully reentrant. The following storage areas are allocated for its processing needs:

Storage Area	Size in Bytes
Working storage	256
Storage required for 1 socket	see the SMARTS environment estimates
Various working storage areas	1024
Storage for the HTTP request processing block	4096
Storage for the receive buffer	RECVBUFL specification
Storage for send buffer	SENDBUFL specification
URL parameter buffer	URLPBUFL specification
Content buffer length	CONTRBUFL specification
For each request header received:	32+1+request header name and header value

Additional Storage Used for CGI Requests

When a CGI request is processed, all of the headers must be set up as environment variables for the CGI program; therefore, the following additional storage is allocated:

Storage Area	Size in Bytes
Environment variable overhead	see the SMARTS environment estimates
Additional data per variable	HTTP request header length + data length + 1

Com-plete Considerations

Any additional storage required by the CGI program must be considered when calculating the thread size and ULIB catalog size required by a given CGI program (in this case, the PAENSTRT program) to function.

Pooled Server Processing

The HTTP server implements pooled server processing dynamically.

When a request is received, the HTTP server checks the pooled server queue to determine if any pool servers are available. For the first request, no pool server is found so the system starts a server to process the request. When the request is processed, the server puts itself on the pooled server processing queue. When the next request arrives, the pooled server is found on the queue and is used to process the request.

Advantages of Pooled Servers

- Using a pooled server rather than starting a new server saves the cost of starting and initializing a new thread of control and terminating the request processing logic. Significant resources are thus saved.

- The system creates pooled servers until a steady state is reached where a pooled server is always available to process an incoming request.
- If a pool server ABENDs for some reason, only the user being processed is affected. If insufficient pooled servers are available, the HTTP server attaches more users.
- Pooled servers are dormant while on the pooled server queue. They use no CPU. In Complete environments, pooled servers are eligible for rollout so they do not occupy a thread.
- The CLEARPOOL operator command makes it possible to clear the pooled server queue and terminate all active pooled servers.

Considerations when Using Pooled Servers

A pooled server can process any type of request. For example, the first request it services may be a static HTML deliver request, the second may be a COBOL program CGI request, the third may be a Natural program CGI request.

- The amount of storage available in the pooled server's thread must be carefully considered to ensure that sufficient unfragmented storage is available to run whatever request may arrive. To that end, it may be advisable to have a separate HTTP server processing Natural requests only.
- Because a pooled server instance never really terminates, CGI programs must clean up after processing to avoid a negative impact on following requests.

Natural Considerations

When a Natural CGI request is issued and a new thread must be started, the Natural interface is started and, using an internal inverted call mechanism, Natural CGI programs are driven.

Using the inverted mechanism to call Natural means that the Natural environment is only started once per pooled server so that the next time a Natural CGI request is handled by the same pooled server, no Natural initialization/termination is required.

In this way, a significant savings is realized in both CPU and wall clock time required to process Natural CGI requests.

Conversational Processing

Instead of maintaining a generic queue of active user threads, a more specific queue is maintained for conversational processing.

A conversation is maintained by sending a session-specific cookie to the browser with the output from the CGI program. The cookie is returned with the next request that enables the HTTP server to match the incoming request with the active request on the conversational request queue.

The incoming data is piped to the program in conversation and the conversational program is dispatched again to process the request from the user with whom it is in conversation.

If a conversational program terminates due to a timeout or an ABEND, the next time the user attempts to converse with the other user in the conversation, a message is issued to indicate that the conversation no longer exists.

Installing NATURAL CGI

SMARTS supports the Natural runtime system as documented in the Natural manuals in the section for Com-plete.

Follow the steps for installing Natural under this environment as required.

The Natural installation should be complete and working in the environments where the HTTP server is to run before attempting to support the Natural CGI processing.

The SMARTS HTTP server requires either Com-plete version 6.1 (or above) or Natural version 3.2 (or above) to support the Natural CGI processing. The Natural Web Interface is supported by Natural version 3.1 and above.

Note:

For a Natural version 2.2 installation under Com-plete, ensure that the HTPvrs.SRCE precedes any Natural datasets in the SYSLIB concatenation in all assembly jobs.

This chapter covers the following topics:

- Natural 2.2 Support
 - Natural Tasks
 - Relationship to the HTTP Server Configuration
 - Invoking a Natural CGI Program
 - Installation Verification
 - Additional Notes
 - Using the Natural Web Interface
-

Natural 2.2 Support

This product has been tested and fully supports Natural 2.2.8 and Natural 2.3.1. Contact Software AG if you need support for earlier, supported versions of Natural version 2.2.

Natural Tasks

A number of additional tasks are required to ensure that the Natural CGI processing operates successfully. Note that the HNANSHEL program delivered with the product is a Natural program itself and an integral part of the Natural CGI processing logic.

Follow the next steps carefully:

1. Ensure that the HTPvrs Natural library has been INPLed as part of the installation process.

2. Copy the HHANSHEL object from the HTPvrs Natural library to the SYSTEM library on FUSER to ensure that this module is always available to the HTTP server.

See step 4 as an alternative to this step when Natural Security is installed.

3. Copy the Natural subprograms USR0050N, USR0330N, and USR1025N from the SYSEXT Natural library to the SYSTEM library on FUSER.
4. As an alternative to step 2 when Natural Security is installed, make SYSEXT a STEPLIB in the Natural environment.

Relationship to the HTTP Server Configuration

The following parameters specified in the HTTP server configuration are directly associated with the use of Natural CGI:

Parameter	Description
HTTPUSER	The default user ID for a user who is able to issue a Natural CGI request. The user ID is passed on to Natural and must therefore be defined, for example, if Natural Security is installed Note: Applies only when running under Com-plete.
LOGON	If the HTTP server is running with security, Natural is passed either the default user ID as specified by HTTPUSER or the user ID for which the user provided a valid user ID and password. Once again, there may be Natural Security or other implications. Note: Applies only when running under Com-plete.
NATLIB	Name of the Natural library where the HTTP server tells Natural to look for a Natural program specified on an HTTP request when no Natural library is specified.
NATPARM	Where appropriate or necessary, Natural parameters may be specified here to override parameters generated in the Natural parameters in the nucleus. The 'STACK' parameter is ignored if specified in this string.
NATTHRD	In the SMARTS server environment, the name of the thread-resident portion of Natural generated during the Natural installation process.

Software AG recommends that you leave the following Natural parameters in the HTTP configuration parameters by default:

IMSG=NO	Prevents Natural from trying to write the 'terminal' if there are certain start-up errors. If not specified, Natural tries to write the message and the NATCGI request ABENDs.
OUTDEST=CONSOLE	Causes Natural to write messages to the console again, where it would otherwise try to send messages to the 'terminal'.

Invoking a Natural CGI Program

A Natural CGI program is invoked using the standard browser URL

`http://ip-addr:port/natcgi/program`

-where 'program' is the Natural program you want to run, which exists in the Natural library specified in the NATLIB configuration parameter.

If you want to provide a Natural library name, use the URL

`http://ip-addr:port/natcgi/library/program`

-where 'library' is the Natural library where 'program' resides.

Note:

All programs to be run using Natural CGI must be cataloged; that is, they must be in object format.

Installation Verification

To test the installation:

1. Invoke a program that does nothing.

Such a program is provided in the HTTPvrs Natural INPL file delivered with SMARTS as HNANWTOP.

2. Stow the program HNANWTOP in your NATCGI library.
3. Try to invoke it from a web browser using the URL

`http://ip-addr:port/natcgi/hnanwtop`

You will receive no output at your browser; however, the message in the program should be written to the operator console.

4. If this is successful, test the sample Natural CGI program:
 - Ensure that the HNANSAMP program is stowed or cataloged in the default CGI library specified by the NATLIB configuration parameter.
 - Invoke the URL

`http://ip-addr:port/httpvrs/srcce/hnannatt.htm`

When you enter your name, the HNANSAMP program should be invoked and echo your name back to the browser.

Additional Notes

The following should also be taken into consideration when installing Natural CGI:

1. The Natural start-up program in all environments must be linked with AMODE=31.
2. There must be no start-up errors from Natural as this will force Natural to try to write to the terminal or CMPRINT about the errors, which in turn can cause unpredictable results. This will also cause the Natural CGI request to fail.

3. If Natural CGI processing cannot be activated for Natural version 2.2, the most likely cause is that the modified NTCOMP macro on the HTPvrs.SRCE dataset has not been used during the compilation of the thread-resident part during the Natural installation process.

Using the Natural Web Interface

Required Tasks

To ensure successful operation of the Natural Web Interface:

1. Ensure that the Natural CGI processing is installed and working correctly; and
2. INPL the modules supplied in the HTPvrs.UPDW update dataset onto the existing SYSWEB library.

The programs NWWAPS and W3APSENV are an integral part of the Natural Web Interface processing.

Invoking a Natural Web Interface Program

To invoke a Natural Web Interface program, use the standard browser URL

`http://ip-addr:port/natcgi/sysweb/nwwaps/library/subprogram`

-where

library	is the Natural library where 'subprogram' resides
subprogram	is the Natural subprogram you want to run

Note:

All programs to be run using the Natural Web Interface must be cataloged; that is, they must be in object format.

Installation Verification

To test the installation, use the demo application supplied with the INPL update:

1. Catalog all the D5* programs in the SYSWEB library using a valid EMPLOYEES DDM.

The demo includes pictures, which are supplied on the HTPvrs.GIFS dataset.

2. To make the pictures accessible, set up an environment variable 'PICTURES' in the CONFIG file for the Com-plete or SMARTS server under which the HTTP server is running.

For example:

`PICTURES=/HTPvrs/GIFS`

3. If required, add the UDB parameter to the Natural start-up parameters in the HTTP server configuration module.

4. Invoke the demo from a web browser using the URL

`http://ip-addr:port/natcgi/sysweb/nwwaps/sysweb/d5menu`

Additional Notes

For additional information about the Natural Web Interface, refer to the Natural 3.1 documentation.

Security

Note:

This chapter applies only when the HTTP server is running in the SMARTS server environment.

Security on the Internet is a major concern of installations publishing IBM mainframe data.

The SMARTS server environment is fully integrated with the Security Authorization Facility (SAF) on OS/390 and MVS/ESA systems and can thus work with CA-ACF/2, RACF or CA-Top Secret. This integration involves

- verifying with the security system any provided user ID and password; and
- building an ACEE for each user that logs on to the system.

The SMARTS server environment then ensures that when the user issues a request to access any resource, the ACEE associated with that request is the one built for the user logging on. In this way, the security system can control access from multiple users with different security profiles from the same address space.

This chapter covers the following topics:

- The Default User
 - HTTP Server Security Integration
 - Natural Security Considerations
 - Implementing SAF Security
-

The Default User

When the HTTP server initially starts processing a request, it knows nothing about the user until it reads various HTTP header areas. Even then, there may be no information about the user.

For this reason, each request is assigned a ‘default’ user ID using the HTTPUSER configuration parameter. The SMARTS server sees the ‘default’ user ID for the duration of the request unless the user has provided some authorization information in the HTTP headers.

When security is active, the ‘default’ user ID receives the CA-ACF/2, RACF or CA-Top Secret authorization of the SMARTS server address space.

To better identify the default user, Software AG recommends that you specify DEFACEE=YES in the configuration parameters. The HTTP server then builds a default ACEE for the user ID specified by HTTPUSER and ensures that each user running with the default user ID runs with an ACEE built for that default user. Note that when DEFACEE is specified, the user ID specified in HTTPUSER must be defined to the security system; otherwise, the HTTP server initialization fails.

HTTP Server Security Integration

The HTTP server integrates with the SMARTS server security facilities fully by passing on to the security system for verification any user ID and password information provided with a HTTP request. If the user ID and password is verified, from that moment on, any request initiated on behalf of that HTTP request is verified based on the user ID provided.

However, because different HTTP servers have different security requirements, it is possible to run the HTTP server in a number of modes as defined by the LOGON configuration parameter.

Note:

SAF security processing must be active in the underlying SMARTS server system before the HTTP server security processing functions correctly.

Logon Allowed (LOGON=ALLOWED)

This is the default mode in which the HTTP server runs. In this mode, all HTTP requests are accepted and dispatched without any logon requirements from the user. They are dispatched with the authorization of the default user as discussed earlier.

If the HTTP request attempts to access a resource to which the default user does not have access, a response is sent to the browser requesting that the user provide authorization information; i.e., a user ID and password. When this is submitted, it is verified with the underlying system and the request may then be repeated using the user's security profile. If the access attempt also fails for security reasons, the user's request is rejected.

This mode is intended for servers where unsecured and secured resources are available. Secured resources are only available to users that provide a valid user ID and password that has access to the secured data. Unsecured resources are available to all users that can connect to the server.

Logon Required (LOGON=REQUIRED)

When this mode is specified, the HTTP server requires a user to provide valid authorization criteria (that is, a valid user ID and password) with each request to the HTTP server. The first time a user attempts an access without authorization information, the HTTP server requests this information from the browser, which in turn requests it from the client.

This information is verified with the underlying security system: access to the server is only permitted if the user ID and password are successfully validated. From that point on, anything that the user does within the server is checked against the security profile for the user ID provided. If a resource is requested to which the user has no access, the request is simply rejected as the user already provided authorization criteria.

This mode is intended for servers where only secure resources are available or servers that should only be used by authorized personnel (that is, by people defined to the security system).

Logon Disallowed (LOGON=DISALLOWED)

When this mode is specified, the HTTP server ignores any authorization information provided as part of the HTTP request. All users connecting to the server run with the authority of the default user. In this way, the default user is only allowed access to that data that should be publicly available on the Internet.

This mode is intended for servers that are available publicly and where the installation does not want user IDs and passwords to be submitted over the net to the server.

HTTP User ID and Password Encryption

When authorization information is provided to an HTTP request, it is encrypted using a simple and publicly available encryption mechanism. It is more secure than TELNET and FTP, which submit passwords in clear text over the net.

The HTTP server security logic fully secures an installation where the network itself is a "trusted" network. Generally, only Intranets may be considered trusted.

Where a server is available publicly on the Internet, a number of additional measures can be used to improve the security offered by this mechanism:

- Force people to change user IDs and passwords on a regular basis.
- Use conversational CGIs. Once the user ID and password are provided on the first CGI request of a conversation, it is no longer necessary to send them again until the conversation terminates.
- Allocate only user IDs and passwords for short periods of time to allow access to the system over a restricted time frame.

The encryption mechanism may also be useful for controlling access to resources that a server provides. While the data may not be sensitive, perhaps only users who have paid for a given service should be able to access the data. The encryption mechanism can ensure that only those who have been supplied with a valid user ID and password can access the system.

Natural Security Considerations

When Natural acquires control with AUTO=ON, the user ID that is active in the SMARTS server environment is supplied to Natural Security. Where no logon has occurred, this is the user ID defined using the HTTPUSER configuration parameter. If a user provides configuration information and this is accepted and verified by the HTTP server, the user ID provided for the logon is passed to Natural Security.

Once again, Natural Security could be set up to restrict the public HTTPUSER user ID while allowing increased access based on a user ID for which a valid security system user ID and password is supplied.

Implementing SAF Security

Security must first be implemented in the SMARTS server environment system by specifying the configuration parameter SECSYS. This parameter is used to inform the SMARTS server environment whether RACF, CA-ACF2 or CA-Top Secret is in place.

Refer to the SMARTS Installation and Operations Manual for more information about configuration parameters.

Once the security system is active, you must determine the appropriate level of access for each HTTP server and set the required LOGON configuration parameter in the server's configuration.

Programming CGI Requests

This chapter covers the following topics:

- HAANUPR: The HTTP Server User Program Request Module
 - HAANCGIG Interface Module
 - HAANCGIL Interface Module
 - HAANCGIP and HAANCGIT Interface Modules
 - CGI Extension Interface Module Status
-

HAANUPR: The HTTP Server User Program Request Module

All current and future user program requests are serviced using the HAANUPR module. The basic call to the module is documented here.

Each call has the format

```
HAANUPR status function parm1 parm2 <...> parmn
```

-where

status	returns the status of the request in the form of a two-byte return code and two-byte reason code in contiguous storage.
function	the name of the function to be invoked. The name must be in character format as described in the following subsections, left aligned in the 16-byte field and padded to the right with blanks.
parm1, parm2, ...	a set of parameters specific to the function named and described in the subsection devoted to that function.

Each available function is described in the following section along with its parameters and a list of return and reason codes and their meanings.

Standard Return and Reason Codes

The following reason and associated return codes may be returned on any request to HAANUPR:

Reason Code	Return Code	Meaning
4	8	Request failed due to insufficient storage. HAANUPR attempts to acquire a save and work area of about 90 bytes from local storage. If it fails, this status is returned.
40	12	Not an HTTP request. The HAANUPR request was issued from a program that was not running as an HTTP server request.
52	12	Unrecognized request. The 16-character function area provided as the second parameter to HAANUPR contained a function request that HAANUPR did not recognize. This may occur for the following reasons: <ul style="list-style-type: none"> • the character string in the field is either misspelled or not a valid function name. • the string contains lowercase characters. All function identifiers must be in uppercase. • the field length was not 16 characters. The function name must be left-aligned and padded to the right with blanks (not nulls).

The CONVERSE Function

The CONVERSE function is used by conversational CGI programs that wish to maintain a connection with the client browser. Refer to the section in this manual that discusses conversational CGI programs for more information. This function may only be issued after an ENABLE-CONVERSE has been successfully issued and some data has been written to stdout. If the CONVERSE is successful, when the program is next dispatched, the user response to the data sent by the CONVERSE will be available to the application program.

This function is invoked as follows:

```
HAANUPR status 'CONVERSE'
```

CONVERSE Parameters

The CONVERSE function has no additional parameters.

CONVERSE Return and Reason Codes

Reason Code	Return Code	Meaning
56	8	Conversational sequence error. The CONVERSE request may only be issued after some data has been written in response to an HTTP request. It is not possible to converse if the user has not received a mechanism with which to respond; i.e., an HTML form.
60	8	Conversation error. A request was received to converse, however, no ENABLE-CONVERSE was previously issued. The CGI program must first indicate that it wishes to establish a conversation by issuing an ENABLE-CONVERSE function call.

The DISABLE-CONVERSE Function

The DISABLE-CONVERSE function indicates that the user program no longer wishes to converse with the client browser. This function may only be issued after an ENABLE- CONVERSE has been successfully issued at some time previously. It must also be issued prior to any output data being sent to the client browser for a given conversation. In other words, after a program has been redispached after a CONVERSE call, if the conversation is to be terminated, the DISABLE-CONVERSE must be issued before any final output is written to the client browser.

This function is invoked as follows:

```
HAANUPR status 'DISABLE-CONVERSE'
```

DISABLE-CONVERSE Parameters

The DISABLE-CONVERSE function has no additional parameters.

DISABLE-CONVERSE Return and Reason Codes

Reason Code	Return Code	Meaning
56	8	Conversational sequence error. The DISABLE-CONVERSE request may only be issued before any data has been written in response to the current HTTP request. The HTTP server must know before the response is written that this is the last output for the conversation.
64	8	Conversations not supported. This indicates that the HTTP server CONV configuration parameter for the server is set to NO indicating that the server is not supporting conversations.

The ENABLE-CONVERSE Function

The ENABLE-CONVERSE function indicates that a CGI program wishes to start a conversation with the client browser. It must be issued before any output whatsoever is issued in response to a request otherwise, the request will fail. Once this request has been issued, the session will remain in conversation with the client browser until a DISABLE-CONVERSE is issued or the program terminates.

This function is invoked as follows:

```
HAANUPR status 'ENABLE-CONVERSE'
```

ENABLE-CONVERSE Parameters

The ENABLE-CONVERSE function has no additional parameters.

ENABLE-CONVERSE Return and Reason Codes

Reason Code	Return Code	Meaning
56	8	Conversational sequence error. The ENABLE-CONVERSE request may only be issued before any data has been written in response to the current HTTP request. The HTTP server must know before the response is written that the CGI program wishes to converse.
64	8	Conversations not supported. This indicates that the HTTP server CONV configuration parameter for the server is set to NO indicating that the server is not supporting conversations.

The GET-DATA Function

The GET-DATA function may be used to get the value of a field name submitted as part of a CGI request. It may also be used to test for the existence of certain fields on the screen which may be used when lists are presented in HTML format to a user. These lists result in a field name with no value being submitted as part of a CGI request when they are selected.

The interface module will search either the input parameter area as provided when using the GET HTTP method, or the content area as provided when using the POST HTTP method. If the variable requested is not found in the input from the HTML page, or if it has not been specified that only the HTML page must be searched, the interface module will check for a server defined variable (i.e. a defined environment variable) of this name. The caller of this interface module does not have to worry about the type of HTTP method that generated the CGI request as this is handled by the interface module.

This function is invoked as follows:

```
HAANUPR status 'GET-DATA' field value length type start
```

GET-DATA Parameters

field	is the name of the field from the HTML page for which this request is being issued. This field name must be terminated with a blank in order for the interface routine to correctly determine the length of the field name for which it is searching. Note: The longest variable name that can currently be handled by this interface is 255 bytes excluding the blank.
value	is a field with a minimum length of the binary value specified in the length field. If this area is smaller than the length specified in the 'length' parameter, overwrites will occur and the results will be unpredictable. When the field name specified in the 'field' parameter is found in the CGI input and has a value associated with it, the value submitted for the field is copied to this area for a maximum length of the length specified in the length parameter. The value is truncated if longer than this and a return and reason code set to indicate this event. If the value is shorter than the length set in the 'length' parameter, the actual length of the value will be set in the 'length' parameter.
length	is a 2-byte binary value containing the length of the area provided for the value to be returned in the 'value' parameter. This is set to the length of the returned value if the field name is found and has a value associated with it.
type	is a one-byte alpha indicating the type of variable which is to be returned and may be used to restrict the search as follows: 'S' Request server defined variable. 'P' Request variable defined on the HTML page. ' ' First found will satisfy request. When the request is completed and the variable requested is found, this field is modified to contain an 'S' or a 'P' depending on where the variable was found.
start	is a two-byte binary field that may be specified to indicate the offset from which the requested variable is to be returned. Its purpose is to enable a program to obtain the contents of a long text field in chunks. When this field is not specified, the default is to start at position 0 of the input field which is the start of the field.

Note:

Where a field name appears twice on a HTML page, only the first is accessible using this mechanism. For this reason, HTML pages designed to work with this mechanism should use unique names although it is perfectly legal in HTML terms to have the same field names specified many times. To process multiple fields with the same name, the LIST-DATA function must be used.

GET-DATA Return and Reason Codes

Reason Code	Return Code	Meaning
8	12	Invalid length supplied in a length field to the interface function.
20	4	Variable length error. It was not possible to return the full length of a variable due to the fact that insufficient space was provided in the users' parameters to hold the value to be returned.
24	12	The format of the parameters provided was invalid.
28	8	Variable name requested was not found in the CGI data.
32	16	A logic error occurred due to the format of the content data provided with the CGI request.
36	8	Insufficient space to return data.
40	12	Returned when these modules are called from a program which is not running as a result of an HTTP request and therefore does not have the data available to satisfy the request.
48	12	Invalid parameter list. This indicates that one or more parameters for a given request have not been provided or contain invalid data.

The LIST-DATA Function

The LIST-DATA function may be used to get a list of both the variable names from the HTML page and their values, and the server defined or environment variables defined at the server for the request. The interface allows that one or more of these variables may be returned at the same time and multiple requests may be made to return all defined variables to the application program.

The server defined or environment variables are returned first, while the variables found in the HTML page are returned once all environment variables have been returned. When issuing multiple requests, the same TOKEN parameter must be provided to the interface each and every time until the list is exhausted.

This function is invoked as follows:

```
HAANUPR status 'LIST' token entries name-length
  name1 value-length1 value1 type1
  name2 value-length2 value2 type2
  <..>
  namen value-lengthn valuen typen
```

LIST-DATA Parameters

token	is a 4-byte binary token used by the interface for multiple requests. When this is null, the listing of variables starts from the first one found. When all variables available cannot be found, this is set to an internal token value to allow the interface to continue the list at the next variable to be returned. The token is reset to null when all available values have been returned to the caller.
entries	is a 4-byte binary field containing the number of variable name and value entries which the application program may accept from the interface in one call. For each entry, a set of return variables (name, value-length, value and type) must be provided, otherwise the results will be unpredictable. When the call completes, this field contains the number of variable sets returned. This must be used in association with the return and reason codes to determine if all data has been returned or if any data has been returned (the last call may have exhausted the list but may not be obvious from the return, reason codes and entries value returned).
name-length	is a 4-byte binary field containing the maximum length of variable name that can be returned. This must be set based on the length of the name fields passed to the interface.

The following constitute a set which is required to return the information about a given variable to the application program. For each entry specified, a set of fields must be provided to contain the data to be returned. Failure to do this will result in unpredictable results.

name	is an alpha field in which the name of a server or HTML page variable is returned. Its maximum length is determined by the name-length variable. If any name to be returned to the application exceeds this length, the value is truncated.
value-length	is a 4-byte binary field containing the maximum length of the value for the variable that can be accepted for this variable instance. The following value field must have at least this amount of space allocated for it, otherwise, overwrites will occur. When a variable value is returned, this field is changed to reflect the true length of the value as determined from the data. If the value is longer than the value specified here, the value is truncated and this field remains unchanged.
value	is an alpha field in which the value for the variable name associated with this value field is returned. It must be at least as long as the value specified in its associated value-length field, otherwise storage overwrites may occur. When a variable is found, its name is returned in the associated 'name' parameter and the value is returned in this field. The actual length of the value is set in the associated value-length field when the variable name is found. If the value is longer than the value-length specification, the value is truncated.
type	is a 1-byte alpha field that indicates where the variable with which it is associated was found. When the variable is an environment variable set in the server environment, this field contains a 'S'. When the variable was found on the HTML page returned from the client, this field contains 'P'.

LIST-DATA Return and Reason Codes

Reason Code	Return Code	Meaning
8	12	Invalid length supplied in a length field to the interface function.
20	4	Variable length error. It was not possible to return the full length of a variable due to the fact that insufficient space was provided in the users' parameters to hold the value to be returned.
24	12	The format of the parameters provided was invalid.
32	16	A logic error occurred due to the format of the content data provided with the CGI request.
36	8	Insufficient space to return data.
40	12	Returned when these modules are called from a program which is not running as a result of an HTTP request and therefore does not have the data available to satisfy the request.
44	4	End of data reached. This will be set for the LIST-DATA function when all data has been returned. The application program should check the 'entries' field as the number of entries returned may be '0' depending on the sequence of LIST-DATA function requests.
48	12	Invalid parameter list. One or more parameters for a given request have not been provided or contain invalid data.

The PUT-BINARY Function

The PUT-BINARY function enables a CGI application program to send output in response to the CGI request. This output is provided to the HTTP server in the same way as 'standard' CGI output is processed.

The PUT-BINARY function differs from the PUT-TEXT function only in terms of the way it processes the parameters passed to it. PUT-BINARY simply takes the data and length provided at face value and passes them directly to the HTTP output processing module. Refer to the section on PUT-TEXT for information about how it processes output.

This function is invoked as follows:

```
HAANUPR status 'PUT-BINARY' data length
```

PUT-BINARY Parameters

data	is the data or the field containing the data to be output in response to the CGI request.
length	is a 2-byte binary value containing the length of the data area provided. The contents of the data area are output for exactly the length specified in this field.

PUT-BINARY Return and Reason Codes

Reason Code	Return Code	Meaning
8	12	Invalid length supplied in a length field to the interface function.
12	4	Warning returned from the HTTP server output processing module.
16	8	Error returned from the HTTP server output processing module.
24	12	The format of the parameters provided was invalid.
32	16	A logic error occurred due to the format of the content data provided with the CGI request.
40	12	Returned when these modules are called from a program that is not running as a result of an HTTP request and therefore does not have the data available to satisfy the request.
48	12	Invalid parameter list. One or more parameters for a given request have not been provided or contain invalid data.

The PUT-TEXT Function

The PUT-TEXT function also enables a CGI application program to send output in response to the CGI request. This output is provided to the HTTP server in the same way as 'standard' CGI output is processed.

The PUT-TEXT function differs from the PUT-BINARY function only in terms of the way it processes the parameters passed to it. PUT-TEXT assumes text output and strips all trailing non-printable characters from the end of the provided data (as determined from the provided data and length) up to the first character that has a value greater than blank. The only exception to this is where a carriage return (X'0D') or a line feed (X'0A') is encountered. In either case, this will also be treated as valid data and treated as the last character in the output data. This is useful where a standard area and length are to be used as output text data as the program generating the output must include the CR or LF characters to format text correctly. Using the PUT-BINARY request, anything following the CR or LF is also treated as data and may cause output to 'skew'.

This function is invoked as follows:

```
HAANUPR status 'PUT-TEXT' data length
```

Note:

Use of this function has no bearing on the way data is translated; only on the way the actual length of the data is calculated prior to output. After it has been output, translation occurs based on the criteria described earlier.

PUT-TEXT Parameters

data	is the data or the field containing the data to be output in response to the CGI request.
length	is a 2-byte binary value containing the length of the data area provided. This must contain the maximum length of the data area. As stated previously, PUT-TEXT strips off all trailing non-printable characters in the data area.

PUT-TEXT Return and Reason Codes

Reason Code	Return Code	Meaning
8	12	Invalid length supplied in a length field to the interface function.
12	4	Warning returned from the HTTP server output processing module.
16	8	Error returned from the HTTP server output processing module.
24	12	The format of the parameters provided was invalid.
32	16	A logic error occurred due to the format of the content data provided with the CGI request.
40	12	Returned when these modules are called from a program which is not running as a result of an HTTP request and therefore does not have the data available to satisfy the request.
48	12	Invalid parameter list. This indicates that one or more parameters for a given request have not been provided or contain invalid data.

HAANCGIG Interface Module

Note:

This documentation is only provided for compatibility. All applications should use the HAANUPR GET-DATA function to achieve this functionality.

The HAANCGIG module obtains the value of a field name submitted as part of a CGI request.

It also determines the existence of fields selected from lists presented to the user in HTML format which result in a field name with no value being submitted as part of a CGI request.

The interface module ascertains the type of HTTP method used to generate the CGI request and searches

- the input parameter area as provided when using the GET HTTP method; or
- the content area as provided when using the POST HTTP method.

If the variable requested is not found in the input from the HTML page and if the search has not been restricted to the HTML page only, the interface module checks for a server-defined variable (that is, a defined environment variable) of this name.

It is not necessary to be concerned about the HTTP method that generated the CGI request when calling the interface module as this is handled by the interface module itself.

The HAANCGIG interface module must be invoked with the following parameter list:

```
HAANCGIG status field value length type start
```

-where

status	returns the status of the request
field	names the field from the HTML page for which the request is being issued. The field name must be terminated with a blank in order for the interface routine to correctly determine the length of the field name for which it is searching. The longest variable name that can currently be handled by this interface is 255 bytes excluding the blank.
value	is a field with a minimum length of the binary value specified in the length field. If this area is smaller than the length specified in the 'length' parameter, overwrites occur and the results are unpredictable. When the field name specified in the 'field' parameter is found in the CGI input and has a value associated with it, the value submitted for the field is copied to this area for a maximum length specified in the length parameter. If the value is longer, it is truncated and a return and reason code are set. If the value is shorter, the actual length of the value is set in the 'length' parameter.
length	is a 2-byte binary value containing the length of the area provided for the value to be returned in the 'value' parameter. This is set to the length of the returned value if the field name is found and has a value associated with it.
type	is a one-byte alpha field indicating the type of variable to be returned. It may be used to restrict the search as follows: 'S' request server-defined variable. 'P' request variable defined on the HTML page ' ' first found satisfies request. When the request is completed and the requested variable is found, this field is modified to contain an 'S' or 'P' depending on where the variable was found.
start	is a two-byte binary field used to indicate the offset from which the requested variable is to be returned. This information makes it possible for a program to obtain the contents of a long text field in chunks. By default, the program starts at the beginning of the field, which is position 0 of the input field.

Note:

If a field name appears twice on an HTML page, only the first occurrence is accessible using this mechanism. Thus HTML pages designed to work with this mechanism should use unique field names, although HTML itself allows the same field names to be specified multiple times.

HAANCGIL Interface Module

Note:

This documentation is provided only for compatibility. All applications should use the HAANUPR LIST-DATA function to achieve this functionality.

The HAANCGIL module is used to obtain a list of

- the variable names from the HTML page and their values; and
- the server-defined or environment variables defined at the server for the request.

The interface allows one or more of these variables to be returned at the same time and multiple requests may be made to return all defined variables to the application program.

The server-defined variables or environment variables are returned first, while the variables found in the HTML page are returned after all environment variables have been returned.

The HAANCGIL interface module must be invoked with the following parameter list:

```
HAANCGIL status token entries name-length
          name1 value-length1 value1 type1
          name2 value-length2 value2 type2
          ...
          namen value-lengthn valuen typen
```

-where

status	is used to return the status of the request as documented in the section Interface Module Status.
token	is a 4-byte binary value used by the interface for multiple requests. When the token value is null, the listing of variables starts from the first one found. When all variables available cannot be found, "token" is set to an internal value to allow the interface to continue the list at the next variable to be returned. The token value is reset to null when all available values have been returned to the caller.
entries	is a 4-byte binary field containing the number of variable name and value entries which the application program may accept from the interface. For each entry, a set of return variables (name, value-length, value, and type) must be provided; otherwise, the results are unpredictable. When the call has completed, this field contains the number of variable sets returned. This number must be used in association with the return and reason codes to determine if all data has been returned or if any data has been returned (the last call may have exhausted the list but may not be obvious from the return/reason codes and entries value returned).
name-length	is a 4-byte binary field that specifies the maximum length of the variable 'name' that can be returned. This length must be set based on the length of the 'namen' fields passed to the interface.
For each entry specified, a set of fields must be provided to contain the data to be returned. Failure to provide these fields produces unpredictable results. The following fields comprise the set required to return the information about a given variable to the application program:	
name	is an alpha field in which the name of a server or HTML page variable is returned. The maximum length of the name is determined by the variable 'name-length'. Any name to be returned to the application that exceeds this length is truncated.

value-length	is a 4-byte binary field containing the maximum length of the variable 'value' that can be accepted for this variable instance. The following 'value' field must have at least this amount of space allocated for it; otherwise, storage overwrites occur. When a variable 'value' is returned, this field is changed to reflect the true length of the 'value' as determined from the data. If the 'value' is longer than the 'value-length' specified here, the 'value' is truncated and the 'value-length' field remains unchanged.
value	is an alpha field in which the value for the variable 'name' associated with this value field is returned. This field must be at least as long as the value specified in the associated 'value-length' field; otherwise, storage overwrites occur. When a variable is found, its name is returned in the associated 'name' parameter and the value is returned in this field. The actual length of the value is set in the associated 'value-length' field when the variable name is found. If the value is longer than the value-length specification, the value is truncated.
type	is a 1-byte alpha field that indicates where the associated variable was found. When the variable is an environment variable set in the server environment, this field contains 'S'. When the variable was found on the HTML page returned from the client, this field contains 'P'.

HAANCGIP and HAANCGIT Interface Modules

Note:

This documentation is provided only for compatibility. All applications should use the HAANUPR PUT-BINARY and PUT-TEXT functions to achieve this functionality.

The HAANCGIP and HAANCGIT modules enable a CGI application program to send output in response to the CGI request. This output is provided to the HTTP server in the same way as 'standard' CGI output is processed. See the section Standard CGI Operation for more information about the processing of this output.

The HAANCGIT interface module differs from the HAANCGIP interface module only in the way it processes the parameters passed to it:

- HAANCGIP accepts the data and length provided at face value and passes them directly to the HTTP output processing module.
- HAANCGIT assumes text output and strips all trailing, nonprintable characters from the end of the provided data (as determined from the provided data and length) up to the first character which has a value greater than blank.

HAANCGIT treats a carriage return (X'0D') or a line feed (X'0A') as valid data and as the last character in the output data. This is useful where a standard area and length are to be used when outputting text data as the program generating the output must include the CR or LF characters to format text correctly. Using the HAANCGIP interface, anything following the CR or LF is also treated as data and may cause output to 'skew'.

The HAANCGIP/T interface modules must be invoked with the following parameter list:

HAANCGIP/T status data length

-where

status	is used to return the status of the request as documented in the section Interface Module Status.
data	is the data or the field containing the data to be output in response to the CGI request.
length	is a 2-byte binary value containing the length of the data area provided. For the HAANCGIP interface, this is the exact length of the data to be sent. For the HAANCGIT interface, this is the maximum length of the data area. As stated previously, HAANCGIT strips off all trailing nonprintable characters in the data area.

CGI Extension Interface Module Status

The status parameter passed to the CGI extension interface modules is a four-byte contiguous area comprising a two-byte return code followed by a two-byte reason code. Appropriate definitions are provided in the language-specific sections; however, the return and reason codes are documented in this section.

The language-related call to the interface modules must be successful before any return or reason codes are entered. The call fails if the interface module

- is not linked with the module produced by the language compiler; or
- could not be loaded at run time if this facility is available to the language.

Interface Module Return Codes

The following codes may be returned by the CGI extension interface modules.

Return Code	The CGI extension interface request . . .
0	was processed successfully
4	was processed successfully but there is additional information related to the calling of the function in the reason code field
8	failed due to some environmental error. The reason code indicates what happened
12	was invalidated based on information the user provided or failed to provide. The reason code indicates what happened.
16	failed due to an internal processing error. The reason code indicates what happened.

Interface Module Reason Codes

Note:

This documentation is provided only for compatibility. All applications should use the HAANUPR interface functions to achieve this functionality.

The following reason codes may be returned by the CGI extension interface modules. Each reason code has an associated return code as documented in this table.

Reason Code	Associated Return Code	Meaning
4	8	Request failed due to insufficient storage
8	12	Invalid length supplied to the interface function in a length field
12	4	Warning returned from the HTTP server output processing module
16	8	Error returned from the HTTP server output processing module
20	4	Variable length error The full length of a variable could not be returned due because insufficient space was provided in parameters supplied by the user to hold the value to be returned.
24	12	The format of the parameters provided was invalid
28	8	The variable name requested was not found in the CGI data
32	16	A logic error occurred due to the format of the content data provided with the CGI request
36	8	Insufficient space to return data
40	12	The module is called from a program that is not running as a result of a HTTP request and therefore does not have the data available to satisfy the request
44	4	End of data reached. This will be set for the HAANCGIL function when all data has been returned. The application program should check the 'entries' field as the number of entries returned may be '0' depending on the sequence of HAANCGIL interface requests.
48	12	Invalid parameter list. One or more parameters for a given request have not been provided or contain invalid data.

Running CGI Programs under SMARTS

This chapter provides information about running application programs under SMARTS in the supported environments.

This chapter covers the following topics:

- The SMARTS Server Environment
 - The Com-plete Environment
 - Natural Considerations
 - C Considerations
 - COBOL Considerations
 - PL/1 Considerations
 - S/390 Assembler Considerations
-

The SMARTS Server Environment

Programs that comply with the HTTP server CGI requirements can use any of the mechanisms available under the SMARTS server environment to access data.

It is currently possible to access ADABAS, DB2, and VSAM data from this environment.

Refer to the appropriate SMARTS documentation for more information about accessing data from this environment.

This section provides information about running programs in the SMARTS server environment.

Linking the Program

An application program may only be run in the SMARTS server environment if it is available to the SMARTS server address space. To make an application program available to the SMARTS server address space, link the program into a dataset in the COMPLIB concatenation under MVS or MSP, or to one of the libraries in the VSE search chain. Always link a reentrant module with the 'RENT' option.

Sample jobs HJBNCBC and HJBNPL1C are provided on the HTPvrs.SRCE dataset to do this once an object module has been created. Creation of the object modules is covered in the appropriate language-specific section of this chapter.

Note:

Since high-level languages like Natural, COBOL, and PL/1 load the SMARTS-provided CGI extensions, COBOL and PL/1 programs must be linked into a dataset in the COMPLIB concatenation whereas no additional processing is required for Natural.

Requirement

All CGI programs must run RMODE=ANY.

The Com-plete Environment

Programs that comply with the HTTP server CGI requirements can use any of the mechanisms available under Com-plete to access data.

It is currently possible to access ADABAS, DB2, and VSAM data from this environment.

Refer to the appropriate Com-plete documentation for more information about accessing data from this environment.

This section provides information about running programs under Com-plete.

Linking the Program for Com-plete

An application program may only be run under Com-plete if it is available to the Com-plete address space. To make an application program available to the Com-plete address space, link the program into a dataset in the COMPLIB concatenation under MVS or MSP, or to one of the libraries in the VSE search chain. Always link a reentrant module with the 'RENT' option.

Sample jobs HJBNCOBC and HJBNPL1C are provided on the HTPvrs.JOBS dataset to do this once an object module has been created. Creation of the object modules is covered in the appropriate language-specific section of this chapter.

Note:

Since high-level languages like Natural, COBOL, and PL/1 load the SMARTS-provided CGI extensions, COBOL and PL/1 programs must be linked into a dataset in the COMPLIB concatenation whereas no additional processing is required for Natural.

Preparing Com-plete for the Application

In general, the user must tell Com-plete about the load module before it will execute successfully in these environments. This process is called 'cataloging' the program and is achieved using the Com-plete ULIB utility. The ULIB utility is provided with the size of the thread below the line in which the program will run. If this calculation is incorrect, the application program is unlikely to run correctly. This is discussed in the next section. Other program-related options may be specified to the ULIB utility as well. Refer to the Com-plete Utilities Manual for more information.

Calculating the Catalog Size under Com-plete

All applications running under Com-plete have two areas of storage available to them: thread storage above and below the 16-megabyte line.

- Storage above the line is the same for all threads and is available in its entirety to any program running in a given thread, regardless of the catalog size specified to the ULIB utility for the program.

- Storage below the line is limited and therefore more tightly controlled. The amount available to an application is controlled by the storage specification set for the program using the ULIB utility.

Note:

The maximum size any given thread can handle is the thread size specified in the Com-plete configuration parameters minus 4k. Thus 496K is the largest application program catalog size that a 500K thread can handle.

The catalog size comprises all the storage the program allocates locally as follows:

- The size of the program if it is loaded into the thread.

The program is loaded into the thread if it is not specified as a Com-plete RESIDENTPAGE program.

- Any working storage allocated by the program.

For languages such as C, C++, COBOL, and PL/1, a table is generally built that indicates the initial values of storage to be allocated. Software AG recommends that you use the sample table provided for batch in each case (only the language environment case if all programs are LE-enabled) and modify it to reflect the storage requirements of the online programs that will run under Com-plete. The language-related documentation or the language environment (LE) documentation tells you how to calculate the storage requirement for any given program.

- Any storage allocated from the thread by the system.

These storage estimates are found in the section Resource Usage.

- Storage for any other programs (not in the RESIDENTPAGE area) called by the main application program and the storage they require from the thread area.

When calculating these values, add 5 to 10% for fragmentation of the storage in the thread area.

Catalog Size for CGI Programs under Com-plete

Because a program that runs as a CGI program is not the first program loaded into the thread, its ULIB definitions are not used to build the thread.

Instead, the ULIB definitions for the PAENSTRT module are used. PAENSTRT must be cataloged large enough to accommodate itself and the CGI programs that it calls.

The storage used by the HTTP server request processing module is documented in the section Resource Usage, while the storage used by the CGI program may be calculated as documented in the previous section.

Program Index Entries under Com-plete

To avoid repeating the same load process every time, Com-plete keeps an index of the most used program names in core. This index maintains information about the program in the load library so that the process of loading the program is quicker.

If the program is relinked, it is generally not found automatically because the index still has a record of the older version of the program. The ULIB utility is used to inform Com-plete that the information about the module must be refreshed.

Com-plete also remembers when a module was not found. If a module was not found and is subsequently added to the load library, you must inform Com-plete that it now exists by refreshing the module in the same way.

Running the Program under Com-plete

After you have logged on through the VTAM interface, you may run the program from the command line of the USTACK screen. Alternately, you may invoke the program by issuing a CGI request for the program to an HTTP server running in the Com-plete environment where the program is available.

Program Options or Functions to Avoid under Com-plete

Compilers offer a number of facilities and options to control the execution of a program. In general, options or functions that impact the following areas should be avoided:

- Abnormal termination recovery or diagnostics processing can impact the data in any subsequent dump and render problem resolution even more difficult. Any dumps or diagnostics provided for support purposes must be generated with these options turned off.
- Use options that load the SMARTS extension modules dynamically at runtime. Otherwise, these modules must be linked to the application program modules, which may cause problems with version control and also makes the module larger. When the language's runtime system loads these modules, they are found in the RESIDENTPAGE area and shared among all application programs and languages.
- Use the SMARTS stdio functions to access sequential operating system files when required. Using I/O functions to access this data will either fail or impact the integrity of the Com-plete system.

Recommendations for the Com-plete Environment

Refer to the Complete System Programmer's Manual for information about using LE/370 with Com-plete.

Software AG recommends that you define the following programs as RESIDENTPAGE:

```
RESIDENTPAGE=CEEBINIT
RESIDENTPAGE=CEEEV005
RESIDENTPAGE=CEEPLPKA
RESIDENTPAGE=IGZCFCC
RESIDENTPAGE=IGZCLNK
RESIDENTPAGE=IGZCPAC
RESIDENTPAGE=IGZCPCO
RESIDENTPAGE=IGZCULE
RESIDENTPAGE=IGZCXFR
RESIDENTPAGE=IGZEINI
RESIDENTPAGE=IGZETRM
RESIDENTPAGE=IGZEVEX
```

Recommendations for Cobol Running under Com-plete

Software AG recommends that you use the following parameters:

```
STACK=( 8K, 8K, BELOW, KEEP ),
STORAGE=( 00, NONE, 00, 8K ),
```

Software AG also recommends that you assemble and link CEEUOPT to the COBOL program.

Natural Considerations

Natural must be installed in the SMARTS server environment before SMARTS functions can be used.

Running Natural Applications

Natural applications that use SMARTS functions are run as normal applications; however, the SMARTS environment must be active and, to run Natural CGI requests, the HTTP server must be active.

Natural and the SMARTS CGI Extensions

To use the SMARTS CGI extensions with Natural

1. INPL it into an appropriate Natural environment.
2. include the local data area HNANCGRL in the HTPvrs Natural library (provided as part of the SMARTS installation materials); and

The HNANCGRL LDA defines the status area and codes that may be returned to the various SMARTS CGI extensions interface requests.

Once HNANCGRL has been included, the Natural program may simply issue calls as follows:

```
CALL 'HAANUPR' HTPCGI-status function parm1 parm2 < .. > parmn
```

The following calls may also be issued but are only included for compatibility with previous releases. The HAANUPR interface module implements the recommended interface: the following interface requests will not be enhanced:

```
CALL 'HAANCGIG' HTPCGI-status field value lengthCALL 'HAANCGIL' HTPCGI-status token entries name-length
    name1 value-length1 value1 type1
    name2 value-length2 value2 type2
    < .. >
    namen value-lengthn valuen typen
CALL 'HAANCGIP' HTPCGI-status data length
CALL 'HAANCGIT' HTPCGI-status data length
```

The parameters required by these interfaces are described in the section SMARTS CGI Extensions . The Natural format required for the other parameters is as follows:

Name	Natural Format	Description
field	A<length>	Must be an alpha field ending in a blank character
variable	A<length>	Must be an alpha field ending in a blank character
value	A<length>	Must be an alpha field of length 'length'
length	I2	Must be a two-byte binary field containing a length
data	no defined format	The start of an area (normally alpha) of length 'length'

Natural Script

Using SMARTS and the Natural ISPF macro facility, you have the option to generate dynamic HTML using the Natural language. The process is known as macro expansion, where the text (or HTML in this case) is generated. This can also consist of variable substitution, repeating blocks, conditionally generated text, along with file I/Os.

The macro facility is an extension of the Natural language and comprises two types of statements: processing statements and text lines. Both are identified by the macro character that is defined by the Natural ISPF administrator.

Processing Statements

Processing statements are executed during the macro expansion. The statements must be preceded by the macro character and followed by a blank. The full power of the Natural language is available for processing statements.

Text Lines

Text lines are copied to the generated output of the macro. They can contain variables or text. A variable must be preceded by the macro character in order to be interpreted during expansion.

Example

In the following example, the macro character is ^. Lines 0010, 0020, and 0040 are processing statements; line 0030 is a text line. Note that the ^ character is in front of the #NAME text line variable without a space between.

Example:

```
0010 ^ MOVE 'DAVE' TO #NAME
0020 ^ IF #NAME EQ 'DAVE' THEN
0030 <b> ^#NAME </b>
0040 ^ END-IF
```

When the example above is expanded, the generated output is as follows:

```
<b> DAVE </b>
```

How It All Works with SMARTS

Although its use is not required, the Natural ISPF macro facility provides a way to simplify the programming of dynamic HTML generation.

Using Natural CGI, you can execute Natural objects directly from the browser. Within the Natural object, calls can be included to get variables from the HTML page the call was initiated from and also to put output back to the browser.

To start, an object of type macro must be created. This object type is only available within Natural ISPF. After the macro is created, Natural code can be added for dynamic generation. The macro objects reside in Natural libraries. When using Natural CGI, all objects must reside in the NATCGI library. Once a Natural ISPF macro is stowed, it can be executed outside of Natural ISPF. When a macro is executed within Natural ISPF, the output is written to the workpool (a Natural ISPF facility). When a macro object is executed in native Natural, the output is written to the editor source area.

To put it all together, the HTML and Natural code are added to the macro object. Once the object has been tested, it is stowed. Within the macro is a call to the HNANCGIP program that uses a Natural routine to read the editor source area and write the lines back to the browser.

Macro Object Example:

```

0010 ^ DEFINE DATA LOCAL
0020 ^ 01 #TITLE (A10)
0030 ^ END-DEFINE
0040 ^ ASSIGN #TITLE = 'This is a Natural Script test'
0050 <HTML>
0060 <HEAD>
0070 <TITLE> ^#TITLE </TITLE>
0080 <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
0090 <META NAME="AUTHOR" CONTENT="A.N. Other">
0100 <META NAME="FORMATTER" CONTENT="Microsoft FrontPage 2.0">
0110 <META NAME="GENERATOR" CONTENT="Mozilla/3.01Gold (WinNT; I) [Netscape]">
0120 </HEAD>
0130 <BODY TEXT="#000000" BGCOLOR="#FFFFFF" LINK="#0000EE" VLINK="#551A8B"
0140 ALINK="#FF0000">
0150
0160 <P><IMG SRC="../images/abc.gif" HEIGHT=28 WIDTH=378><BR>
0170 <A HREF="/imagemap/barmenu.map"><IMG ISMAP SRC="../images/BARMENU.gif"
0180 BORDER=0 HEIGHT=28 WIDTH=378></A></P>
0190
0200 <H2>This is a test </H2>
0210 </BODY>
0220 </HTML>
0230 ^ FETCH RETURN 'HNANCGIP'
```

The only Natural lines in this example are on 0010, 0020, 0030, 0040, and 0230. These are all processing statements. The rest of the lines are all text lines. Note that the variable #TITLE is preceded by the macro character on line 0070. During the macro expansion, this variable is substituted with the actual value of the variable.

The line 0230 is calling a Natural program called HNANCGIP. As mentioned earlier, the output is written to the editor source area when macros are executed outside of Natural ISPF. HNANCGIP reads all lines in the editor source area and puts them back to the browser. It is that easy to create dynamic HTML. When you want to change the HTML, you can create new HTML using a tool such as FrontPage and then just cut and paste it into the macro. Then insert the Natural code for dynamic generation, stow the object, and

test it.

After the macro is expanded, the output written to the editor source area looks like the following:

```
0010 <HTML>
0020 <HEAD>
0030 <TITLE> This is a Natural Script test </TITLE>
0040 <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
0050 <META NAME="AUTHOR" CONTENT="A.N. Other">
0060 <META NAME="FORMATTER" CONTENT="Microsoft FrontPage 2.0">
0070 <META NAME="GENERATOR" CONTENT="Mozilla/3.01Gold (WinNT; I) [Netscape]">
0080 </HEAD>
0090 <BODY TEXT="#000000" BGCOLOR="#FFFFFF" LINK="#0000EE" VLINK="#551A8B"
0100 ALINK="#FF0000">
0110
0120 <P><IMG SRC="../images/abc.gif" HEIGHT=28 WIDTH=378><BR>
0130 <A HREF="/imagemap/barmenu.map"><IMG ISMAP SRC="../images/BARMENU.gif"
0140 BORDER=0 HEIGHT=28 WIDTH=378></A></P>
0150
0160 <H2>This is a test </H2>
0170 </BODY>
0180 </HTML>
```

Note that the actual value of #TITLE was substituted on line 0030. When the output is in the editor source area, HNANCGIP reads all lines and puts them to the browser.

The output written to the browser looks like the following:

```
<HTML>
<HEAD>
<TITLE> This is a Natural Script test </TITLE>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
<META NAME="AUTHOR" CONTENT="A.N. Other">
<META NAME="FORMATTER" CONTENT="Microsoft FrontPage 2.0">
<META NAME="GENERATOR" CONTENT="Mozilla/3.01Gold (WinNT; I) [Netscape]">
</HEAD>
<BODY TEXT="#000000" BGCOLOR="#FFFFFF" LINK="#0000EE" VLINK="#551A8B"
  ALINK="#FF0000">

<P><IMG SRC="../images/abc.gif" HEIGHT=28 WIDTH=378><BR>
<A HREF="/imagemap/barmenu.map"><IMG ISMAP SRC="../images/BARMENU.gif"
BORDER=0 HEIGHT=28 WIDTH=378></A></P>

<H2>This is a test </H2>
</BODY>
</HTML>
```

Additional Notes on Natural

The following additional hints and tips may prevent problems while using the SMARTS CGI extensions with Natural:

1. Do not present the first element of a Natural data area structure as a parameter to these routines. Natural will present each element of the structure as a single parameter rather than simply passing a pointer to the structure. Allocate the entire area, redefine the field, and name the main variable as the parameter to the function.

2. Certain functions can result in system ABENDs that terminate the Natural session. Normally, this only happens when certain functions are used in an invalid fashion but are not trapped by Natural abnormal termination routines.

C Considerations

The SMARTS Software Developer Kit (SDK) is required to support CGI programs based on C.

Compiling and Linking C Applications

C application programs must be linked using the standard process that would be used to compile a batch program for the operating system where the program will run. The one difference is that the APSvrs.SRCE dataset provided with the SMARTS SDK must be specified instead of the C header library provided with the compiler or operating system.

As SMARTS SDK does not provide or support all of the 800 or so UNIX function interfaces available at present, there may be occasions when it is possible to use the header files and implementation provided with the operating system. Whether this works depends on the function and how closely it interacts with the operating system. Software AG recommends that you first write and run a small program under SMARTS SDK to determine if the functionality will cause a problem.

It is intended that SMARTS SDK will support most if not all of these interfaces in the future. If you need support for a function interface that is not currently supported, contact your Software AG technical support representative for information about when and how the function will be supported. Refer to the SMARTS SDK Programmer's Reference Manual for information about the functions that are currently supported.

Supplied C Sample Programs and Jobs

The C samples for HTTP processing that are provided on the HTPvrs.SRCE dataset are described in the following table.

Member	This is a sample ...
HCANSAMP	C program to accept a simple CGI request and return some data to the user. When compiled and linked, it may be used in conjunction with the source members HHANCGET or HHANCPUT, which contain HTML and are provided on the HTPvrs.SRCE dataset. Note: PJBSCC and associated link jobs may be used to compile and link this sample if required.
HHANCGET	HTML page that drives the HCANSAMP C program using a form and the HTTP GET method. It may be referenced using the following URL. Refer to the installation verification procedure for information about interpreting this URL. http://ip-addr:port/htpvrs/srcce/hhancget.htm
HHANCPUT	HTML page that drives the HCANSAMP C program using a form and the HTTP POST method. It may be referenced using the following URL. Refer to the installation verification procedure for information about interpreting this URL. http://ip-addr:port/htpvrs/srcce/hhancput.htm

SMARTS and stdin, stdout, and stderr

The standard I/O files are all supported by SMARTS as follows:

File	Standard Processing	CGI Processing
stdin	stdin is empty and any attempt to access it sets the end-of-file condition.	When the request was generated using the POST method, stdin contains the content data for the CGI request. When the GET method is used, stdin is empty and returns end-of-file, if accessed.
stdout	Output to stdout is written to the 'stdout' DD/DLBL which is allocated either explicitly in the SMARTS procedure or by default as a spool file.	Output to stdout is taken as response data to the CGI request and passed on to the requesting client.
stderr	Output to stderr is written to the 'stderr' DD/DLBL which is allocated either explicitly in the SMARTS procedure or by default as a spool file.	Same as for standard processing

C and the SMARTS CGI Extensions

The SMARTS CGI extensions were designed with non-C applications in mind. It is not envisaged that C application programs will use these extensions.

COBOL Considerations

COBOL programs must be compiled and linked as if for the batch environment on the operating system where they will run. Once compiled and available to the SMARTS address space or partition, COBOL programs may simply be run as documented earlier in this manual.

Sample Programs and Jobs

The following table describes the various HTTP server COBOL samples that are provided on the HTPvrs.SRCE dataset.

Member	This is a sample ...
HHANCOBT	HTML page that drives the HOANSAMP COBOL program using a form and the HTTP GET method. It may be referenced using the following URL. Refer to the installation verification procedure for more information about interpreting this URL. http://your.ip.address:port/httpvrs/srcce/hhancobt.htm
HJBNCOBC	JCL member to compile and link the sample COBOL CGI program HOANSAMP.
HOANCONV	COBOL program that uses the conversational features of the HTTP server to enable it to converse with a WWW browser over a series of HTML pages. It may be started using the following URL. Refer to the installation verification procedure for more information about interpreting this URL. http://ip-addr:port/cgi/hoanconv
HOANSAMP	COBOL CGI program that is driven by the HHANCOBT HTML page.

COBOL and the SMARTS CGI Extensions

COBOL programs may use the SMARTS CGI extensions by issuing a call to the appropriate extension module as follows:

```
call 'HAANUPR' using HTPCGI-STATUS, function, parm1, parm2, <...>, parmn.
```

The following calls may also be issued but are only included for compatibility with previous releases. The HAANUPR interface module implements the recommended interface; the following interface requests will not be enhanced:

```
call 'HAANCGIG' using HTPCGI-STATUS, field, length, value.
call 'HAANCGIL' using HTPCGI-status, token entries, name-length,
    name1, value-length1, value1, type1,
    name2, value-length2, value2, type2,
    <...>
    namen, value-lengthn, valuen, type2.
call 'HAANCGIP' using HTPCGI-STATUS, data, length.
call 'HAANCGIT' using HTPCGI-STATUS, data, length.
```

The parameters required by these interfaces are described in SMARTS CGI Extensions .

The HTPCGI-status must be defined as follows in COBOL:

```
01 HTPCGI-STATUS.
   03 HTPCGI-RETURN-CODE pic 9(4)COMP.
   03 HTPCGI-REASON-CODE pic 9(4)COMP.
```

The following describes the COBOL format required for the other parameters:

Name	COBOL Format	Description
field	pic x(<length>)	Must be an alpha field ending in a blank character
variable	pic x(<length>)	Must be an alpha field ending in a blank character
value	pic x(<length>)	Must be an alpha field of length 'length'
length	pic 9(4) COMP	Must be a two-byte binary field containing a length
data	no defined format	The start of an area (normally alpha) of length 'length'

PL/1 Considerations

PL/1 programs must be compiled and linked as if for the batch environment on the operating system where they will be running. Once compiled and available to the SMARTS address space or partition, PL/1 programs may be run as documented earlier in this manual.

Sample Programs and Jobs

The following table describes the HTTP server PL/1 samples that are provided on the HTPvrs.SRCE dataset.

Member	This is a sample ...
HHANTPL1T	HTML page that drives the HPANSAMP PL/1 program using a form and the HTTP GET method. It may be referenced using the following URL. Refer to the installation verification procedure for more information about interpreting this URL. http://ip-addr:port/httpvrs/srcce/hhancget.htm
HJBNPLIC	JCL to compile and link the sample PL/1 CGI program HPANSAMP.
HPANSAMP	PL/1 CGI program that is driven by the HHANPL1T HTML page.

PL/1 and External Module Names

Because PL/1 cannot support external names longer than seven (7) characters, aliases are required for the SMARTS extension interfaces. All future interface names will be at most seven (7) characters long.

Aliases are created by changing the first four (4) characters of any extension module name to PL1. Users must create these aliases themselves if they wish to use the functionality.

The following table provides a cross reference between the extension program name as documented in the SMARTS SDK Programmer's Guide and what must be used by PL/1:

PAANATOE	PL1ATOE
PAANETOA	PL1ETOA

These external modules must also be declared to the PL/1 program as follows:

```
DCL PAANHLL EXTERNAL ENTRY OPTIONS(ASM INTER);
DCL PL1ATOE EXTERNAL ENTRY OPTIONS(ASM INTER);
DCL PL1ETOA EXTERNAL ENTRY OPTIONS(ASM INTER);
DCL HAANUPR EXTERNAL ENTRY OPTIONS(ASM INTER);
```

Finally, for PL/1 to load these program dynamically, the following statements must be included:

```
FETCH PAANHLL;
FETCH PL1ATOE;
FETCH PL1ETOA;
FETCH HAANUPR;
```

If these statements are not included, the PL/1 compiler expects them to be linked with the application program, which is not recommended.

PL/1 and the SMARTS CGI Extensions

PL/1 programs may use the SMARTS CGI extensions by issuing a call to the appropriate extension module as follows:

```
CALL 'HAANUPR' using HTPCGI-status, function, parm1, parm2, <...>, parmn.
```

The calls documented in the other sections have not be included here as no PL/1 applications have been created with older versions of SMARTS.

The parameters required by this interface is described in the section SMARTS CGI Extensions .

The HTPCGI-status must be defined as follows in PL/1:

```
DCL 1 HTPCGI-STATUS,
    2 HTPCGI-RETURN-CODE FIXED BINARY(15),
    2 HTPCGI-REASON-CODE FIXED BINARY(15);
```

The following describes the PL/1 format required for the other parameters:

Name	PL/1 Format	Description
field	CHAR(<length>)	Must be an alpha field ending in a blank character
variable	CHAR(<length>)	Must be an alpha field ending in a blank character
value	CHAR(<length>)	Must be an alpha field of length 'length'
length	FIXED BINARY(15)	Must be a two-byte binary field containing a length
data	no defined format	The start of an area (normally alpha) of length 'length'

S/390 Assembler Considerations

Assembler and the SMARTS CGI Extensions

Software AG recommends that Assembler programs use the standard SMARTS API to handle CGI requests.

Support and Maintenance

The SMARTS system nucleus is written entirely in IBM 390 Assembler and is therefore supported using ZAPs as the quickest and easiest way to provide corrections to customers.

This chapter covers the following topics:

- Reporting Problems
 - Problem Resolution
 - Applying Maintenance
-

Reporting Problems

Problems should be reported to your local technical support center. You will be asked to provide whatever information is required to solve the problem. Generally, you should have the following available when reporting a problem:

1. Version, revision, and SM level of the SMARTS HTTP server software where the problem occurred.
2. Type and level of operating system where SMARTS was running.
3. Version, revision, and SM level of other products associated with the problem (for example, Natural, ADABAS).
4. Message numbers where applicable.
5. System log for a period of time before the event.
6. Sequence of actions used to cause the problem, if reproducible.
7. Name and offset of the module where the problem occurred. Where an ABEND occurs within a SMARTS module, RC generally points to the start of the module where you will find a constant identifying the module. Subtract the PSW address from the address in RC to provide the offset into the module.
8. The register contents at the time of the ABEND.

With this information, it may be possible to identify a previous occurrence of the problem and a correction. If this is not the case, the following additional information is required:

1. The Com-plete online dump or SMARTS address space dump, as appropriate.
2. Output from the job where the failure occurred.
3. Other information that support personnel feel is relevant.

Problem Resolution

A number of tools are available to diagnose HTTP server problems as follows.

Thread Dump Diagnosis under Com-plete

When an application program ABENDs within the SMARTS environment running under Com-plete, an online dump is written to the SD file. This dump may be viewed immediately and online using the UDUMP utility. Refer to the Com-plete Utilities Manual for more information.

The dumps may be printed using the batch utility TUDUMP. Refer to the Com-plete System Programmer's Manual for more information about TUDUMP.

HTTP Server Trace Facilities

When HTTP requests are not being processed successfully, it can be useful to trace incoming and outgoing data. For incoming data, the HEADER trace will provide details of exactly how HTTP server has interpreted a given request while DATA tracing will show the exact format of data as it is received at the HTTP server side and what is actually sent back by the HTTP server request processing module (or the CGI program) to the web browser. HTTP server tracing may be activated using the HTTP server TRACE configuration parameter.

Applying Maintenance

ZAPs for problems in the SMARTS product are provided in the following format:

HTvrnnn

-where

HT	identifies this as a ZAP for the SMARTS HTTP server
vr	is the version and revision number of SMARTS to which the ZAP applies
nnn	is a sequential number uniquely identifying the ZAP

When a ZAP is provided to correct a problem, Software AG recommends that you use the following procedure:

1. Copy the load modules zapped by the fix to a temporary load library.
2. Apply the ZAP to the modules in the temporary load library using the AMASPZAP utility.

Note:

When a ZAP applies to an environment-specific module (that is, one beginning with the characters HAeN or PAeN where "e" is any character other than "A"), it may be necessary to relink the module to activate the change.

3. Run SMARTS, placing this temporary load library in front of the standard HTPvrs.LOAD dataset in the COMPLIB concatenation.

4. Ensure that the problem has been resolved. If this is not possible immediately, it may be advisable to run in this way for a period of time until it is clear that
 - the ZAP has not caused any problems; and
 - the problem the ZAP is intended to fix has been corrected.
5. If the ZAP causes problems or does not clear the problem, the temporary load library may be deleted or cleared.
6. When you have verified the correction, copy the zapped modules back into your HTPvrs.LOAD dataset.

The HTTP Server User Exit

The single HTTP server user exit is given control at strategic points in the processing of each HTTP request to enable an installation to control and manipulate the processing of HTTP requests.

This chapter covers the following topics:

- Installation
 - General Interface
 - Exit Points
-

Installation

The HTTP server user exit must be a load module called HAANUXIT and must be available for loading by the HTTP server.

It is loaded when the HTTP server is initialized and becomes part of the HTTP server nucleus. It must be reentrant and may reside above the 16-megabyte line.

The member HAANUXIT on the HTPvrs.SRCE dataset is a sample user exit that implements each of the user exit points currently taken but simply returns normally to the user exit point. Thus, the user exit as delivered may be compiled and will not affect the normal operation of the HTTP server.

A message is issued during the server initialization process when the module is found and loaded. If this message is not issued, the exit is not operational.

General Interface

The exit is called with the following registers:

Reg.	Contents
0	No relevant data
1	Pointer to a parameter list as described in this section
2-12	No relevant data
13	Pointer to an 18F savearea, which may be used by the exit
14	Address to which the exit returns. The exit must return using the following instruction to insure that the caller gets control back in the appropriate mode: BSM R0,R14
15	Pointer to the entry point of the module

Exit Parameter List

On entry to the user exit, register 1 points to a parameter list that varies depending on the call.

The parameters passed to each exit point are documented in the appropriate section; however, the first parameter is always a pointer to the UXIT control block, which contains information about the HTTP server itself and the exit point for which the user exit has been called.

A DSECT to map this control block may be generated by the following statement in the Assembler program:

```
MYUXIT    HMANUXIT DSECT=YES
```

The parameter list may be addressed using the following statements assuming R1 contains what it contained when the module was entered:

```

L        Rx,0(,R1)           Rx -> UXIT control block
USING    UXIT,Rx             Address UXIT area

```

The following table describes the field names and their contents, which are relevant for all exit points. The UXIT control block may also contain data relevant only to specific exit calls. Any fields of additional significance are documented in the description of the exit point itself.

Field Name	Description
UXITFUNC	Contains a function code indicating the exit point for which the exit has been called.
UXITSSNM	Names the HTTP server subsystem for which the exit is being called. Each HTTP server has a subsystem name assigned when the server is started. This field allows the exit to uniquely identify an HTTP server when more than one is running in the same SMARTS address space or partition.
UXITUSER	A field available to the user. This will be NULL on entry to the exit the first time and will be passed intact to the user for each subsequent call. This field is intended for use as an anchor point for the exit where storage areas are maintained from call to call. In particular, the exit must insure that any storage areas acquired during the exit processing and returned to any given exit point are freed during the termination call.

The HAANUXIT source member on the HTPvrs.SRCE dataset shows how the UXIT parameter list may be addressed and processed. It should be used as the basis for any exit written at an installation.

Entry/Exit Processing

The HTTP server has been written using a standard entry/exit mechanism to provide consistent standards within the server and to centralize much of the common coding used during entry/exit processing.

Two macros are used:

- HMANENT for entry processing; and
- HMANEXIT for exit processing.

HMANENT Macro for Entry Processing

The HMANENT macro takes a number of parameters; however, the important ones from the exit's point of view are the WRKL and WRKD parameters.

- WRKL specifies the length of the savearea and work area that should be allocated for the exit.
- WRKD specifies the name of the working storage DSECT that will be used to address the area.

When control is passed to the code following the HMANENT macro expansion, the following registers are set:

Reg.	Contents
0	As passed by the calling program (for HAANUXIT, contains no relevant data)
1	As passed by the calling program (for HAANUXIT, contains a pointer to the HAANUXIT parameter list)
2-11	As passed by the calling program (for HAANUXIT, contains no relevant data)
12	Set up as the program base register
13	Pointer to a newly allocated savearea and work area allocated based on the length specified on the WRKL parameter. The DSECT name provided on the WRKD parameter must have the following format: WORK DSECT WORKSAVE DS 18F Exit work fields here WORKL EQU *-WORK Note that the field names are not important in this case once they are provided correctly on the HMANENT macro. The main point is that the area must start with an 18F savearea.
14	Address to which the exit returns. While the exit returns using the HMANEXIT macro, this may be useful for determining the mode of the calling program. In the case of the HAANUXIT, it is always called from programs running AMODE=31.
15	Pointer to the entry point of the module

HMANEXIT Macro for Exit Processing

The HMANEXIT macro returns control to the calling program, optionally returning values in registers 15, 0, and 1. For HAANUXIT, the only relevant register in this case is 15, which is used as a return code.

Note that the HMANEXIT macro must get control with registers 12 and 13 containing the same values as were set up by the HMANENT macro.

SMARTS API

The SMARTS Software Developer Kit (SDK) is required to make the full SMARTS API available to the user exit. These functions may be invoked as documented in the SMARTS SDK Programmer's Guide.

Exit Points

For each exit point, the following subsections will be documented:

Subsection	Documents . . .
Purpose	the purpose for which the exit point is intended.
Parameters	the parameters, if any, supplied at a given exit point in addition to the UXIT control block.
Return Codes / Return Values	the return codes that may be returned in register 15 and the processing that any given return code causes. In all cases when register 15 contains an invalid return code on return from the exit, processing continues as described for return code 0.

Initialization

Purpose

The initialization exit point is designed to provide an opportunity for the exit to initialize its own environment prior to any subsequent calls.

Parameters

Parm	Description
1	Pointer to the UXIT control block

Return Codes / Return Values

Return	Description
0	Continue processing normally

Termination

Purpose

The termination entry point has been designed to provide the exit with a point at which it should clean up after any given request.

Parameters

Parm	Description
1	Pointer to the UXIT control block

Return Codes / Return Values

Return	Description
0	Continue processing normally

URL Processing

Purpose

The URL processing exit point is taken immediately before the URL as provided by the user is processed. The installation can then

- modify or replace the URL all together and cause the HTTP server to use a different URL to service the request. This may be useful when URL names change and a cut-over period is required to handle old and new requests.
- reject access to the URL, which causes the HTTP server to return a 'permission denied' response to the request.

Parameters

Parm	Description
1	Pointer to the UXIT control block
2	Pointer to the URL. This is a null1-terminated field. If this area is modified as it stands, the URL processed is the one returned in this area. If the pointer is replaced, this is not be used unless an appropriate return code is set on return from the exit.
3	Pointer to a fullword containing the length of the URL excluding the null-termination byte at the end. If the address in parameter 2 is replaced, this field must be updated to reflect the length of the URL in the area to which parameter 2 now points.

Return Codes / Return Values

Return	Description
0	Continue processing normally. The URL is located and returned as provided to the exit or as modified in place by the exit.
4	The second parameter contains the address of a new storage location containing the URL to be used. The fullword pointed to by the third parameter has been updated to reflect the length of this new URL.
8	Access to the URL is denied. The request is rejected with a 'permission denied' response.

Output Processing

Purpose

The output processing exit point that is taken one or more times as output is built to be sent in response to the request. How often the exit is taken depends totally on what the URL represents and how large the output object is.

This exit point may be used to

- control translation of the output data if the basic facilities provided by the HTTP server are not adequate. This exit can effectively replace the HTTP server translation processing and do the translation itself.
- insert data into the output data stream. This is useful for inserting HTTP headers for any given request; however, users must be careful that the resulting output is still a valid HTTP request. The HTTP server does not control what is being sent in this case and therefore the user is responsible for ensuring that it is correct.
- prevent certain data from being sent.

Parameters

Parm	Description
1	Pointer to the UXIT control block Two additional flags are used in this control block as follows: UXITMHHS => HTTP header sent. UXITMCHS => Content-type header sent. UXITMCON => Now sending HTTP content.
2	Pointer to output data. All data sent in response to a request is passed to this exit starting with the HTTP protocol headers and finishing with the content for the request, if any. If the data area pointed to by this pointer is modified in any way, the modified value is used for 'normal' processing. If this pointer is replaced, the data is only sent if an appropriate return code is returned from the exit.
3	Pointer to a fullword containing the length of the data pointed to by parameter 2. This must be modified if the length of data in the area pointed to by parameter 2 changes or the address is changed to point to a new area.
4	Pointer to the content type being sent. This reflects the content type as the HTTP server believes it to be. If the exit changes the content type in any way, it should be reflected here by the exit. When a CGI or Natural CGI request is being processing, this contains the value 'CGI' or 'NATCGI' until the content type being sent by the CGI program is established by the program itself sending a 'CONTENT-TYPE' header or a default header being sent by the HTTP server itself.

Return Codes / Return Values

Return	Description
0	Continue processing normally. Normal processing means that the data in the area pointed to by parameter 2 for a length based on the fullword pointed to by parameter 3 is sent in response to the request. If headers are still being processed (that is, UXITMCNS is not on), these are always translated from EBCDIC to ASCII. If content is being processed (that is, UXITMCNS is on), the data is only translated from EBCDIC to ASCII if the CONTENT-TYPE header is of the form TEXT/* (where '*' is any value).
4	Do not translate the data. The data is sent exactly as is.
8	Do not send the data. The attempt to send the data is ignored.
12	Insert the data pointed to by a parameter 2 for a length specified by the fullword pointed to by parameter 3. The data is translated based on the rules for return code 0. The next call to the exit represents the same data as was presented on this call, thus enabling the exit to insert HTTP headers and/or content data if desired.
16	Same as return code 12; however, no HTTP server translation is performed on the data.

Input Processing

Purpose

Called when input is being processed for an incoming HTTP request, the input processing exit point may be used to

- translate incoming content in a way different from the basic translation services provided by the HTTP server.
- track or report identified HTTP headers or requests identified by certain requests.

Parameters

Parm	Description
1	Pointer to the UXIT control block
2	Pointer to the data relating to the header identified by parameter 4
3	Length of the data pointed to by parameter 2
4	Pointer to the name of the HTTP header in EBCDIC for which the data pointed to by parameter 2 has been provided. Two 'logical headers' are defined for this interface ('logical' because they do not exist in the HTTP protocol specification and are used only to identify the data): QUERY_STRING the parameter string on the URL CONTENT the actual request content
5	Pointer to a fullword containing the length of the HTTP header pointed to by parameter 4.
6	Points to the CONTENT-TYPE for the request when it is identified.
7	Points to a fullword containing the length of the content type as pointed to by parameter 6.

Return Codes / Return Values

Return	Description
0	Continue processing normally. 'Normal processing' translates any HTTP header data from ASCII to EBCDIC unless it contains encoded data. Content is translated if the incoming content type is TEXT/* (where '*' represents any value).
4	Do not translate: use the data exactly as is in the buffer.

Accept Processing**Purpose**

The accept processing exit point is taken immediately after a client has connected to the HTTP server. It may be used to control which IP addresses may connect with the HTTP server

Parameters

Parm	Description
1	Pointer to the UXIT control block
2	Pointer to a fullword containing the address of the peer application which has connected to the server. This is in the form of an Internet address as follows: H Address family: always '2' to indicate an Internet address H Port number of the peer program F IP address of the peer program This structure is set to nulls when the address of the peer application cannot be established. Note: Where a proxy is in use, this is the IP address of the proxy and not of the originating client.

Return Codes / Return Values

Return	Description
0	Continue processing normally; that is, accept and process the request.
4	No current meaning.
8	Reject the request. The HTTP server immediately terminates the connection with the client.

Messages and Codes

This chapter contains messages and codes issued by the HTTP server components of the SMARTS system.

The explanation of the messages is organized as follows:

- SMARTS HTTP ABEND Codes
- HTTP Server Messages (APSHTTP Prefix)

Message Format

SMARTS HTTP server messages have the following format:

APSHTPnnnn - message-id

—where

HTP	identifies the HTTP subsystem as issuing the message.
nnnn	is a sequential message number identifying the message within the subsystem.
message-id	identifies the SMARTS address space issuing the message. This value is determined from the SMARTS server environment configuration parameter MESSAGE-ID. It may be a single character in braces (e.g. (X)) or it may contain the installation ID as specified in the SMARTS server environment configuration parameter INSTALLATION. Refer to the <i>SMARTS Installation and Operations Manual</i> for more information.

Message Documentation

Messages are documented with the message identifier (excluding the constant ‘APS’ which is always present) in the heading followed by the message text.

The message text appears something like the following:

This is a test &1 with three (&2) replacement parms '&3'

When displayed or written to the console, the message contains the text as displayed; however, the placeholders identified by the ‘&n’ construct are replaced by data relevant to the message.

If the data to be displayed is X, y, and AB, the message appears as follows:

This is a test X with three (y) replacement parms 'AB'

Each message includes the following subsections:

Description	explains why the message was issued.
Placeholders	describes what each of the placeholders (that is, the '&n' values) contains. If the message contains no placeholders, the expression 'Not applicable' is written.
Action	describes the actions to be taken when this message is issued. A number of actions may be listed, if appropriate. If no action is required, the expression 'Not applicable' is written.
Additional References	describes any other additional source(s) of information that may further explain the message. A number of references may be listed, if appropriate. If no additional references are available, the expression 'Not applicable' is written.

SMARTS HTTP ABEND Codes

Overview of Messages

ABEND U2000 | ABEND U2001 | ABEND U2002 | ABEND U2003

ABEND U2000

Insufficient Storage

Explanation

ABEND U2000 occurs when insufficient storage is available to obtain a primary control block within the SMARTS HTTP environment. It indicates a problem at a point where it is not possible to issue a SMARTS HTTP error message.

Action

Make sufficient storage available to the program running in the environment to enable the request to be satisfied.

ABEND U2001

Logic Error in SMARTS HTTP Nucleus

Explanation

ABEND U2001 indicates an unexpected condition within the nucleus for which a message could not be issued.

Action

It could occur due to storage overwrites in the SMARTS HTTP environment or as a result of earlier errors in the environment. If none of these appears to be the case, report the problem to your local support center.

ABEND U2002

Invalid SMARTS local storage stack area

Explanation

On freeing the SMARTS HTTP server request's local storage stack area, it was found to be corrupt.

Action

See below for storage corruption explanation.

ABEND U2003 Storage Corrupted

Explanation While freeing storage within the SMARTS nucleus, a corruption failure was detected.

Action When storage is allocated, SMARTS puts a storage accounting prefix at the start of the storage and an identical suffix at the end of the storage. If these do not match when the storage is freed, this ABEND occurs.

A mismatch indicates that either

- the user that allocated the storage requested "n" bytes but used more than "n" bytes thus overwriting the storage accounting area at the end; or
- a user of storage before the storage area being freed overwrote the prefix of the storage area.

In any event, register 7 will point to the prefix accounting area of the storage. The length of the storage is found at register 7 + 4 (if it hasn't been corrupted) while the storage area itself returned to the user can be found at register 7 + 8.

HTTP Server Messages (APSHTP Prefix)

Overview of Messages

HTP0001 | HTP0002 | HTP0003 | HTP0004 | HTP0005 | HTP0006 | HTP0007 | HTP0008 | HTP0009 |
 HTP0010 | HTP0011 | HTP0012 | HTP0013 | HTP0014 | HTP0015 | HTP0016 | HTP0017 | HTP0018 |
 HTP0020 | HTP0021 | HTP0022 | HTP0023 | HTP0024 | HTP0026 | HTP0027 | HTP0028 | HTP0030 |
 HTP0031 | HTP0032 | HTP0033 | HTP0034 | HTP0035 | HTP0036 | HTP0037 | HTP0039 | HTP0040 |
 HTP0041 | HTP0042 | HTP0043 | HTP0044 | HTP0047 | HTP0048 | HTP0054 | HTP0055 | HTP0056 |
 HTP0057

HTP0001 HTTP server already active

Explanation An attempt was made to issue an INIT command to an HTTP server; however, the HTTP server was already active.

Placeholder Not applicable

Action Not applicable

References Not applicable

HTP0002 Insufficient storage for \$2 (\$1 bytes)

Explanation An attempt by the HTTP server to acquire storage failed due to insufficient storage either in a user program thread or in the SMARTS address space.

Placeholders

\$1	Number of bytes that the system tried to acquire. The number may be suffixed with a 'K' or an 'M' to denote KBYTES or MBYTES, respectively. '0' indicates that the HTTP server was not in a position to determine the amount of storage that could not be acquired. This could occur if a request to an underlying system failed due to a storage shortage without indicating how much was required.
\$2	A character string indicating what the storage was for and possibly a four-digit code in braces indicating which control block this storage was for, if applicable.

Action

- If the storage is thread-related, use the ULIB utility to increase the catalog size of the HTTP server application program that suffered the error. Note that the thread sizes in general may have to be increased in the SMARTS region depending on how much thread space the application requires.
- If the storage is outside of the thread, there is a shortage of storage in the SMARTS region itself. Where possible, SMARTS will expand its storage areas so it is likely that such an expansion request failed thus resulting in this message. Check for other errors related to any attempted expansion for more details.
- The following table identifies the storage areas by name, where they may be located and what they are used for:

Storage name	Location	Description
Master control block (HMCB)	Above	The main control block used by the HTTP server; acquired from the SMARTS environment general buffer pool.
Working storage	Above	The generic term used for storage acquired on a temporary basis. It will normally be allocated in the HTTP server application program thread.
HTTP request areas (HPRQ and buffers)	Above	The main HTTP server request processing block and associated buffers acquired by the HTTP server request processing program HAANRQST from the application program thread.
HTTP request header (HPRH)	Above	The request header control block acquired per request header found in any given HTTP request. It is allocated from the HTTP server request processing program (HAANRQST) thread.
HTTP environment variable work area	Above	Area allocated from the HTTP server request program thread storage to issue 'putenv' requests to the SMARTS environment to build environment variables for CGI requests.

HTP0003 Invalid parameter specified \$1

Explanation An invalid parameter was specified on the HTTP server startup. This may have been supplied on the SMARTS server environment SERVER configuration parameter or operator command.

Placeholders	<table><tr><td>\$1</td><td>The string presented as a parameter to the HTTP server, which was not recognized.</td></tr></table>	\$1	The string presented as a parameter to the HTTP server, which was not recognized.
\$1	The string presented as a parameter to the HTTP server, which was not recognized.		

- Action**
- Correct the SMARTS server environment configuration parameter SERVER to start the HTTP server.
 - Reissue the SMARTS server environment operator command SERVER INIT with a valid parameter value.

References SMARTS Installation and Operations Manual

HTP0004 Server initialization failed

Explanation An attempt to start an HTTP server failed. The reason for the failure is indicated in a previously issued message.

Placeholders Not applicable

Action Correct the problem that caused the HTTP initialization failure as specified in the preceding message.

References Not applicable

HTP0005 URL parameter data exceeds maximum (\$1 Bytes)

Explanation A user issued an HTTP request with a parameter on the URL that exceeded the length of the URL parameter area. The URL parameter is the data following the question mark, which in turn follows the actual URL itself. The URL parameter buffer is allocated based on the HTTP server configuration parameter URLPBUFL.

Placeholders	<table><tr><td>\$1</td><td>The maximum URL parameter length of data in bytes that is acceptable to the instance of the HTTP server on which the error occurred.</td></tr></table>	\$1	The maximum URL parameter length of data in bytes that is acceptable to the instance of the HTTP server on which the error occurred.
\$1	The maximum URL parameter length of data in bytes that is acceptable to the instance of the HTTP server on which the error occurred.		

Action Determine if the request itself is valid and if so, increase the specification of the HTTP server configuration parameter URLPBUFL accordingly.

References The chapter HTTP Server Use and Customization starting on page in this manual.

HTP0006 Content length exceeds maximum (\$1 Bytes)

Explanation A user issued an HTTP request with content that exceeded the length of the allocated content area. The content of an HTTP request is the data that follows the last HTTP header provided for a request. The content buffer is allocated based on the specification of the HTTP server configuration parameter CONTBUFL.

Placeholders

\$1	The maximum length of content data in bytes which is acceptable to the instance of the HTTP server on which the error occurred.
-----	---

Action Determine if the request itself is valid and, if so, increase the specification of the HTTP server configuration parameter CONTBUFL accordingly.

References The chapter HTTP Server Use and Customization starting on page in this manual.

HTP0007 Content length required for request

Explanation Where content is provided for an HTTP request, a 'CONTENT-LENGTH' header must be provided on the request to determine how much content to expect. This message is issued when the last HTTP header is received that is followed by content data; however, no content length header has been received.

Placeholders Not applicable

Action Determine which client is submitting the HTTP request and change it to either submit no content data or to submit a content length header indicating how much content data will follow.

References The HTTP V1/1 Protocol specification

HTP0008 Server is not active

Explanation An operator command has been issued to an HTTP server or an attempt has been made to terminate an HTTP server that is not active.

Placeholders Not applicable

Action Not applicable

References Not applicable

HTP0009 Content length ‘\$1’ invalid for request

Explanation Content data has been provided on a HTTP request. Content data has been detected and a content length header has been provided; however, the content data provided was less than or greater than the length specified in the content length header.

Placeholders	\$1 The length of data expected based on the length provided in the content length HTTP header.
---------------------	--

Action Determine which client issued the HTTP request and correct its content length processing.

References The HTTP V1/1 Protocol specification

HTP0010 URL ‘\$1’ not found

Explanation A client requested that the URL indicated in placeholder \$1 be returned; however, the URL does not exist.

Placeholders	\$1 The URL as requested by the client program.
---------------------	--

Action This generally occurs for a dataset or dataset and member request. The HTTP server translates a URL of the form /a/b/c/ to a sequential dataset name of the form a.b.c , or a URL of the form /x/y/z to a PDS member name of the form x.y(z). The sequential dataset or PDS member as translated from the URL by the HTTP server was not found on the system. This indicates that either it doesn't exist or it has not been cataloged.

References Not applicable

HTP0011 Unable to load CGI program ‘\$1’

Explanation A request was issued that resulted in the URL being interpreted as a CGI request for the program identified by placeholder \$1. This program was not available to the SMARTS address space and therefore the CGI request could not be completed.

Placeholders	\$1 Name of the program identified by the CGI request which could not be loaded in the SMARTS region where the request was received.
---------------------	---

Action Determine if the request was valid and if it was, make the program available in the SMARTS region where the request should be processed.

References The chapter Running CGI Programs under SMARTS starting on page in this manual.

HTP0012 Natural startup module ‘\$1’ not found: ‘\$2’ NATCGI request failed

Explanation A request was received by the HTTP server that resulted in an attempt to execute the Natural CGI program identified by the \$2 placeholder. It was not possible to process this request as the Natural thread-resident portion identified by the \$1 placeholder could not be loaded in the SMARTS region where the request was received.

Placeholders

\$1	Name of the thread-resident Natural portion that is identified on the HTTP server configuration parameter NATTHRD.
\$2	Name of the Natural program that the Natural CGI request was attempting to execute.

Action Correct the specification of the HTTP configuration parameter NATTHRD to identify the correct Natural thread portion; or make the Natural thread portion named available to the SMARTS region where the request was received.

References The chapter Installing Natural CGI starting on page in this manual.

HTP0013 Module \$1 loaded

Explanation The module identified by the \$1 placeholder was loaded by the nucleus. This message is issued in the following cases:

- When more than one version of a module exists, it indicates which version of the module was loaded.
- For exits that may not normally be part of the nucleus, this indicates when an exit has been loaded and is active in the system.

Placeholders

\$1	Name of the module that was loaded.
-----	-------------------------------------

Action This is an informational message, no action is necessary.

References Not applicable

HTP0014 Server initialization in progress

Explanation The HTTP server has commenced its initialization processing.

Placeholders Not applicable

Action Not applicable

References Not applicable

HTP0015 Read error ‘\$1’ processing URL ‘\$2’

Explanation A URL identifying a sequential dataset or PDS member has been received. This dataset or PDS member has been found; however, an error occurred while reading the data to transmit it to the user.

Placeholders

\$1	The error number returned by the SMARTS API function being used to read the dataset or PDS member. Refer to the <i>SMARTS SDK Programmer's Reference Manual</i> for a cross reference of error numbers to C macro variable names identifying the error that has occurred. The HTTP server uses the ‘gets’ and ‘read’ functions to access data, so refer to the descriptions of these functions for details of the cause of the identified error number being returned.
\$2	The URL as requested by the user.

Action Use the returned information to identify the problem and correct it.

References SMARTS SDK Programmer's Guide SMARTS SDK Programmer's Reference Manual

HTP0016 ‘putenv’ failed processing environment variables errno=\$1

Explanation When a CGI request is identified, the HTTP server must make a number of specific environment variables available to the CGI program. This is done by using the SMARTS API ‘putenv’ function to set the environment variables before calling the CGI program. This indicates that the function request failed and therefore one or more environment variables will not be available to the CGI program called.

Placeholders

\$1	The error number as returned by the ‘putenv’ function. Refer to the <i>SMARTS SDK Programmer's Reference Manual</i> for details of errors returned by the ‘putenv’ function.
-----	--

Action Based on the information returned, determine why the ‘putenv’ function failed and correct the problem. Normally, ‘putenv’ only fails when insufficient storage is available in the thread where the CGI program runs. This is determined by the size at which the HTTP server request processing module (default name HAANRQST) is catalogued using the ULIB utility.

References SMARTS SDK Programmer's Reference Manual

HTP0017 \$1 Sockets \$2 request error errno=\$3

Explanation The HTTP server uses the SMARTS API sockets functions to communicate with clients. This message indicates that the request indicated by the \$1 placeholder failed with the error number indicated.

Placeholders	\$1	Name of the sockets function request that failed
	\$2	Error number
	\$3	Error number

Action Sockets errors generally occur when the peer program terminates the conversation, or a transport error occurs on the network.

References SMARTS SDK Programmer's Reference Manual

HTP0018 Access to URL '\$1' forbidden

Explanation A user attempted to access the URL identified by \$1 but access to the resource was denied. This occurs depending on the specification of the HTTP server configuration parameter LOGON as follows:

- When LOGON=REQUIRED is specified, or LOGON=ALLOWED is specified and the user supplies authorization information (user ID and password) for the request, this error indicates that access to the resource based on the provided information was not granted.
- When LOGON=DISALLOWED is specified and the 'public' user ID does not have access to the requested resource, the user is not permitted to provide authorization information so the request is refused.

Placeholders	\$1	The URL that the user attempted to access.
---------------------	-----	--

Action If the resource should be accessible to users of the HTTP server in question, grant the necessary access rights. If not, determine why users are attempting to access the resource in question.

References The information on Security.

HTP0020 Internal logic error in \$1+x\$2

Explanation Something unexpected has occurred in the HTTP server nucleus. Additional errors are likely to follow any occurrence of this error message.

Placeholders	\$1	Name of the HTTP server nucleus program in which the error was detected.
	\$2	Hexadecimal offset in the program where the error was detected.

Action Report the error message and the steps taken to produce the error to your local support center.

References Not applicable

HTP0021 Server is quiescing

Explanation The HTTP server is quiescing. In this state, current requests may continue to completion; however, no new requests are allowed.

Placeholders Not applicable

Action This is an informational message: no action is necessary.

References Not applicable

HTP0022 SAF Logon for default userid \$1 failed

Explanation The default user ID specified is not authorized.

Placeholders	\$1	Default user ID
---------------------	-----	-----------------

Action Check that the default user ID specified in the HMANCONF macro by the parameter HTTPUSER is valid.

References The information on Security

HTP0023 Invalid URL '\$1'

Explanation The URL requested by a client is invalid.

Placeholders	\$1	The URL as requested by the client
---------------------	-----	------------------------------------

Action Specify a URL that the HTTP Server can understand. Refer to the appropriate IETF standards for information on what constitutes a valid URL.

References Not applicable

HTP0024 Natural initialization failure, RC=\$1

Explanation A request was received by the HTTP server that resulted in an attempt to execute a Natural CGI program. The Natural thread-resident portion that is identified on the HTTP server configuration parameter NATTRD was loaded, but an error occurred at startup.

Placeholders	\$1 The return code received from the Natural thread-resident portion.
---------------------	---

Action Determine from the Natural documentation why the return code is returned and correct the error.

References Natural Installation and Operations Manual.

HTP0026 Conversation \$1 no longer exists

Explanation When a conversation is started with a WWW browser, it is assigned an internal identifier known only to the HTTP server and the browser with which it is conversing. This error indicates that when the browser eventually sent data in response to a request, the original conversation no longer existed. This could occur due to a timeout, a system ABEND, or even a stored URL being sent minutes, hours, days, or even weeks after the conversation was terminated normally by the user.

Placeholders	\$1 Name of the conversational cookie.
---------------------	---

Action Restart the conversational application as appropriate.

References Not applicable

HTP0027 Input buffer space \$1 bytes exhausted

Explanation This error occurs if pooled sessions are in use and the entire request from the user must fit into the input buffer. The message is longer than the input buffer length specified in the configuration for the HTTP server.

Placeholders	\$1 The length of the buffer used for receiving input data from the network.
---------------------	---

Action Increase the specification of the HTTP server configuration parameter RECVBUFL.

References The chapter HTTP Server Use and Customization starting on page in this manual.

HTP0028 Server interface \$1 request failed rc=\$2

Explanation A call to the environment independent server interface failed.

Placeholders	\$1	The server function requested: initialization, termination, command
	\$2	Return code

Action One or more messages will be written to the console to indicate the nature of the error. Correct these problems and restart the server.

References Not applicable

HTP0030 HTTP header '\$1' unrecognized

Explanation A header was detected in an HTTP request that was not recognized by the HTTP server. The header is ignored and the request is processed normally, ignoring the specified header.

Placeholders	\$1	Name of the header found in the HTTP request that was not recognized by the HTTP server.
---------------------	-----	--

Action Report this to your technical support representative. While this is not an error as such, it is possible that the HTTP header should be recognized in a future release of the HTTP server.

References Not applicable

HTP0031 Unsupported request data \$1/x\$2

Explanation In the unformatted portion of an HTTP request, the HTTP server expects data as specified in the HTTP V1/1 protocol standard. If more information than normal is provided, or if the HTTP server makes an error interpreting the request, this message is issued. The request is processed normally; however, the data indicated in the message placeholders is be taken into account in the processing of the request.

Placeholders	\$1	The data provided on the request that the HTTP server ignored in character format.
	\$2	The data provided on the request that the HTTP server ignored in hexadecimal format.

Action Report this to your technical support representative. While this is not an error as such, it is possible that the HTTP request data should be recognized in a future release of the HTTP server.

References Not applicable

HTP0032 Header '\$1' data '\$2' ignored

Explanation The HTTP server recognized the header identified by the \$1 placeholder, however, it had to ignore the data as identified by the \$2 placeholder. This generally occurs due to a shortage of working space in the HTTP server request processing areas. The request is processed, however, the ignored header data, as shown in this message, is not taken into account while processing the request.

Placeholders	\$1	Name of the HTTP header for which the data was ignored.
	\$2	The data (or at least the start of the data) that was ignored for the header identified by \$1.

Action Report this to your technical support representative. While this is not an error as such, it indicates that certain values within the HTTP server request interpretation processing may need to be reviewed.

References Not applicable

HTP0033 Environment interface \$1 request failed rc=\$2

Explanation A call to the environment dependent interface failed. Environmental components are called for initialization, termination and for initialization of the listening task.

Placeholders	\$1	The server function requested: ENVINIT, ENVTERM, or LISTINIT
	\$2	Return code

Action Report the error to your local support center.

References Not applicable

HTP0034 \$1 Invalid data \$2/x\$3 for program

Explanation One of the HTTP server thread processing programs HAANLIST or HAANRQST is started by a mechanism apart from the internal processing of the HTTP server. These programs expect specific parameters that can only be provided via the internal HTTP server mechanisms.

Placeholders	\$1	Name of the program that was started with the invalid data.
	\$2	The invalid data provided to the program in character format.
	\$3	The invalid data provided to the program in hexadecimal format.

Action Determine who or what is attempting to start these programs in an incorrect way and force them to stop.

References Not applicable

HTP0035 SMARTS API \$1 Request failed rc=\$2/errno=\$3

Explanation An error occurred for a request made to the SMARTS environment.

Placeholders	\$1	Name of the request.
	\$2	Return code.
	\$3	Error number.

Action Refer to the SMARTS SDK Programmer's Reference Manual to determine why the errno was returned for the request and correct the error. Note that in many cases, these "errors" may be normal due to general activity on the TCP/IP network, for example.

References SMARTS SDK Programmer's Guide

HTP0036 Server waiting for \$1 user(s) to terminate

Explanation The server cannot terminate correctly until all users have terminated. The message indicates the number of users upon which the server is waiting.

Placeholders	\$1	The number of users still active.
---------------------	-----	-----------------------------------

Action Wait until all users have terminated and reissue the request to QUIESCE or TERMINATE the server. The server may be forced; however, this is not recommended due to the subsequent problems it can cause.

References Not applicable

HTP0037 Operator command \$1 issued successfully

Explanation The operator command identified in the message has been successfully issued to the server.

Placeholders	\$1	Name of the operator command issued.
---------------------	-----	--------------------------------------

Action This is an informational message: no action is necessary.

References Not applicable

HTP0039 HTTP server \$1 active on port \$2

Explanation The HTTP listening task attached for the server indicated by the \$1 placeholder has been started successfully and is now listening for incoming HTTP requests on the TCP/IP port identified by the \$2 placeholder.

Placeholders	\$1	Name of the HTTP server as specified on the SMARTS server environment SERVER configuration parameter or operator command.
	\$2	The TCP/IP port number on which the HTTP server is listening. This is set using the HTTP server configuration parameter PORT.

Action Not applicable

References The HTTP Server installation ionformation.

HTP0040 Module not found \$1

Explanation The module identified by the \$1 placeholder cannot be found. A request to the operating system to load the module has failed. Modules to be loaded by the HTTP server must be available either in

- the COMPLIB DD concatenation or system LNKLIST for OS/390 and MSP systems; or
- a library identified in the search path for VSE.

If this message is issued during the initialization process, initialization fails if the module is required for the correct operation of the HTTP server. Otherwise, initialization continues.

If this message is issued during the termination process, termination continues; however, depending on the function of the module, the termination process may not complete successfully.

Placeholders

\$1	Name of the module that could not be found
-----	--

Action If the module should be available during initialization and/or termination processing, determine why it cannot be found.

References Not applicable

HTP0041 Module \$1 load error rc=\$2/\$3

Explanation The module identified by the \$1 placeholder could not be loaded due to an error during LOAD processing. A request to the operating system to load the module failed for some reason other than the fact that the module could not be found.

If this message is issued during the initialization process, initialization fails if the module is required for the correct operation of the HTTP server. Otherwise, initialization continues.

If this message is issued during the termination process, termination continues; however, depending on the function of the module, the termination process may not complete successfully.

Placeholders	\$1	Name of the module for which the LOAD request failed.
	\$2	Return code from the operating system LOAD request.
	\$3	Reason code from the operating system LOAD request.

Action Determine from the return and reason codes why the LOAD request failed and correct the error.

- References** • *MVS/ESA Assembler Programmers Macro Reference Manual*
- *VSE/ESA Assembler Programmers Macro Reference Manual*

HTP0042 Module \$1 issued return code \$2

Explanation A number of modules are called internally during HTTP server initialization and termination. These modules generally issue a 0 return code when they complete successfully. This message is issued when a module is called and its return code is not 0. When a non-zero return code is issued, the module responsible issues a message itself to indicate where the problem lies.

Initialization processing continues if the return code is less than 8 and terminates if the return code is 8 or greater.

Termination processing continues; however, if the return code is 8 or greater, there may be additional failures later in the termination process.

Placeholders	\$1	Name of the module that issued the return code
	\$2	Return code issued by the module identified by \$1.

Action Refer to preceding messages in the log to determine why the return code was issued. Correct the situation.

References Not applicable

HTP0043 Server \$1 initialization successful

Explanation Initialization processing for the HTTP server identified by the \$1 placeholder completed successfully.

Placeholders	\$1	Name of the HTTP server as specified on the SMARTS server environment SERVER configuration parameter or operator command.

Action Not applicable

References Not applicable

HTP0044 Server \$1 terminating on port \$2

Explanation The HTTP server as identified in the \$1 placeholder is no longer listening for incoming HTTP requests on the port specified by placeholder \$2. This generally indicates that an error has occurred on the sockets 'listen' or 'accept' request, which in turn indicates that the TCP/IP connection has problems.

Placeholders	\$1	Name of the HTTP server as specified on the SMARTS server environment SERVER configuration parameter or operator command.
	\$2	The TCP/IP port number on which the HTTP server was listening.

Action Not applicable

References SMARTS Installation and Operations Manual

HTP0047 Server \$1 terminated

Explanation The HTTP server identified by the \$1 placeholder terminated successfully.

Placeholders	\$1	Name of the HTTP server as specified on the SMARTS server environment SERVER configuration parameter or operator command.

Action Not applicable

References Not applicable

HTP0048 Unrecognized operator command \$1

Explanation An operator command was issued to the HTTP server using the SMARTS server environment SERVER operator command; however, the operator command was not processed because it was not recognized by the HTTP server.

Placeholders	<table border="1"> <tr> <td data-bbox="389 373 438 495">\$1</td><td data-bbox="438 373 1383 495">The operator command provided on the SMARTS server environment SERVER operator command for the HTTP server that was not recognized by the HTTP server.</td></tr> </table>	\$1	The operator command provided on the SMARTS server environment SERVER operator command for the HTTP server that was not recognized by the HTTP server.
\$1	The operator command provided on the SMARTS server environment SERVER operator command for the HTTP server that was not recognized by the HTTP server.		

Action Issue a valid operator command.

References The chapter HTTP Server Use and Customization starting on page in this manual.

HTP0054 Server \$1 using \$2 configuration

Explanation The HTTP server identified by the \$1 placeholder is initializing with the HTTP server configuration parameters specified by the \$2 placeholder.

Placeholders	\$1	Name of the HTTP server as specified on the SMARTS server environment SERVER configuration parameter or operator command.
	\$2	Name of the HTTP server configuration parameter module that will be used by this instance of the HTTP server for its configuration information.

Action Not applicable

References The chapter HTTP Server Use and Customization starting on page in this manual.

HTP0055 Invalid configuration file \$1 data=\$2/x\$3

Explanation The HTTP server configuration file identified by the \$1 placeholder starts with invalid data and thus cannot be used by the HTTP server for its configuration processing. The HTTP server initialization process terminates.

Placeholders	\$1	Name of the configuration file found to be invalid.
	\$2	The data found at the start of the invalid module in character format.
	\$3	The data found at the start of the invalid module in hexadecimal format.

- Ensure that the module was generated using the procedures and samples provided with SMARTS.
- Ensure that another module of the same name is not higher in the COMPLIB concatenation or VSE search path than the HTTP configuration module you wish to use.

References The information on installation, as well as customization and use of the HTTP Server.

HTP0056 Server cannot initialize - SMARTS environment not active

Explanation The SMARTS environment must be active before the HTTP server can be activated. During its initialization processing, the HTTP server determined that the SMARTS environment was not active.

Placeholders Not applicable

Action

- If the HTTP server is being started with the SMARTS environment SERVER configuration parameter, ensure that it appears before the HTTP SERVER statement(s) in the SMARTS environment configuration file.
- If the HTTP server is being started with the SMARTS environment SERVER operator command, start the SMARTS environment first.

References *SMARTS Installation and Operations Manual*

HTP0057 Module '\$1' attach failed rc=\$2

Explanation The HTTP server uses internal SMARTS server environment functions to start new processes in the SMARTS region. An attempt to attach the program identified by the \$1 placeholder failed.

If this program is HAANLIST, no task was available to listen on the appropriate port for a given instance of the HTTP server.

If the module name is HAANRQST (or an alternate name assigned by the user), that particular request itself fails; however, the HTTP server remains active and listening

Placeholders

\$1	Name of the program for which the attach request failed
\$2	Return code from the SMARTS server environment request issued as follows:

4	Insufficient TIBs available in the system
8	Program to be attached was not found
12	Security error
16	Invalid program name to be attached
20	Insufficient space in the SMARTS server environment general buffer pool

RC	Action
4	The number of TIBs available in the system is determined by the SMARTS server environment configuration parameter TIBTAB. Review the parameter setting based on the appropriate section in the <i>SMARTS Installation and Operations Manual</i> .
8	The program cannot be found if the HTTP server configuration parameter HTTPRQST specifies a program name that is not available to the SMARTS system.
8, 16	Report these errors to your technical support representative.

Action

RC	Action
4	The number of TIBs available in the system is determined by the SMARTS server environment configuration parameter TIBTAB. Review the parameter setting based on the appropriate section in the <i>SMARTS Installation and Operations Manual</i> .
8	The program cannot be found if the HTTP server configuration parameter HTTPRQST specifies a program name that is not available to the SMARTS system.
8, 16	Report these errors to your technical support representative.

References

- *SMARTS Installation and Operations Manual*
- *Com-plete System Programmer's Manual*

Sample Members

Note:

Refer to the installation verification procedure for more information about interpreting the URLs provided in the following tables.

This document covers the following topics:

- On the SMARTS Source Dataset
 - On the SMARTS HTTP JOBS Dataset
 - On the Natural Library HTPvrs
 - On the Natural INPL Update for SYSWEB
-

On the SMARTS Source Dataset

The following section lists the sample members on the source dataset and tells what they are used for.

Member	Description
HAANCONF	the HTTP server configuration member.
HAANDSNT	source used to map the dataset name to content type for the HTTP server.
HAANTYPT	source used to map the member type to content type for the HTTP server. Can be modified and reassembled to add, remove, or change entries in the table.
HAANUXIT	user exit to alter the processing of requests by the HTTP server.
HCANSAMP	C program to accept a simple CGI request and return some data to the user. When compiled and linked, it can be used in conjunction with the source members HHANCGET or HHANCPUT on the HTPvrs.SRCE dataset, which contain HTML. Note: The PJASCC and associated link jobs may be used to compile and link this sample, if required.
HHANCGET	HTML page that drives the C program HCANSAMP using a form and the HTTP GET method. The page may be referenced as: http://ip-addr:port/htpvrs/srce/hhancget.htm
HHANCOBT	HTML page that drives the COBOL program HOANSAMP using a form and the HTTP GET method. The page may be referenced as: http://your.ip.address:port/htpvrs/srce/hhancobt.htm
HHANCPUT	HTML page that drives the C program HCANSAMP using a form and the HTTP POST method. The page may be referenced as: http://ip-addr:port/htpvrs/srce/hhancput.htm
HHANNATT	HTML page that drives the Natural program HNANSAMP using a form and the HTTP GET method. The page may be referenced as: http://ip-addr:port/htpvrs/srce/hhannatt.htm
HHANPL1T	HTML page that drives the PL/1 program HNANSAMP using a form and the HTTP GET method. The page may be referenced as: http://ip-addr:port/htpvrs/srce/hhancget.htm
HOANCONV	COBOL program that uses the HTTP server to converse with a web browser over a series of HTML pages. The program is started using http://ip-addr:port/cgi/hoanconv
HOANSAMP	COBOL CGI program driven by the HTML page HHANCOBT.
HPANSAMP	PL/1 CGI program driven by the HTML page HHANPL1T.

On the SMARTS HTTP JOBS Dataset

The following section lists the sample members on the jobs dataset and tells what they are used for.

Member	Description
HJBNA CNF	job to compile and link the HAANCONF configuration parameters.
HJBNC OBC	job to compile and link the COBOL CGI program HOANSAMP.
HJBND SNT	job to compile and link the HAANDSNT configuration table.
HJBNP LIC	job to compile and link the PL/1 CGI program HPANSAMP.
HJBNT YPT	job to compile and link the HAANTYPT configuration table
HJBNU XIT	job to compile and link the HAANUXIT user exit.
HJENL INK	job to link the HTTP server extensions for the SMARTS server environment.
HJENP ARM	the parameters used to start and run the HTTP servers under the SMARTS server environment.

On the Natural Library HTTPrs

Member	Description
HNANCGIP	Natural program used for Natural script, which builds an output HTML page in the Natural source area and writes it out to a browser using the HTTP server extensions.
HNANPGDA	Global data area (GDA) used by HNANCGIP.
HAANSAMP	Natural CGI program that is launched by HTML page HHANNATT.
HNANSHEL	Natural CGI interface shell program delivered in object format.
HNANCGRL	Local data area (LDA) containing definitions of the SMARTS API high-level language interface return and reason codes.
HNANW TOP	Natural program that writes to the operator and is used to verify that Natural CGI processing is operating correctly.

On the Natural INPL Update for SYSWEB

Member	Description
NWWAPS	Natural Web Interface extension program delivered in object format
W3APSENV	Natural subroutine required by NWWAPS delivered in object format
D5*	27 Natural modules (subprogram/subroutine/copycode/parameter/text) delivered in source format, which make up the Natural Web Interface demo application

